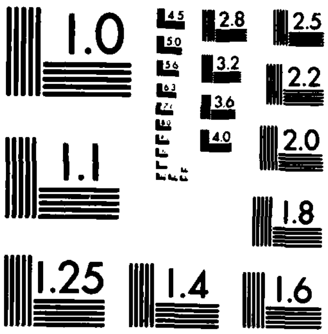
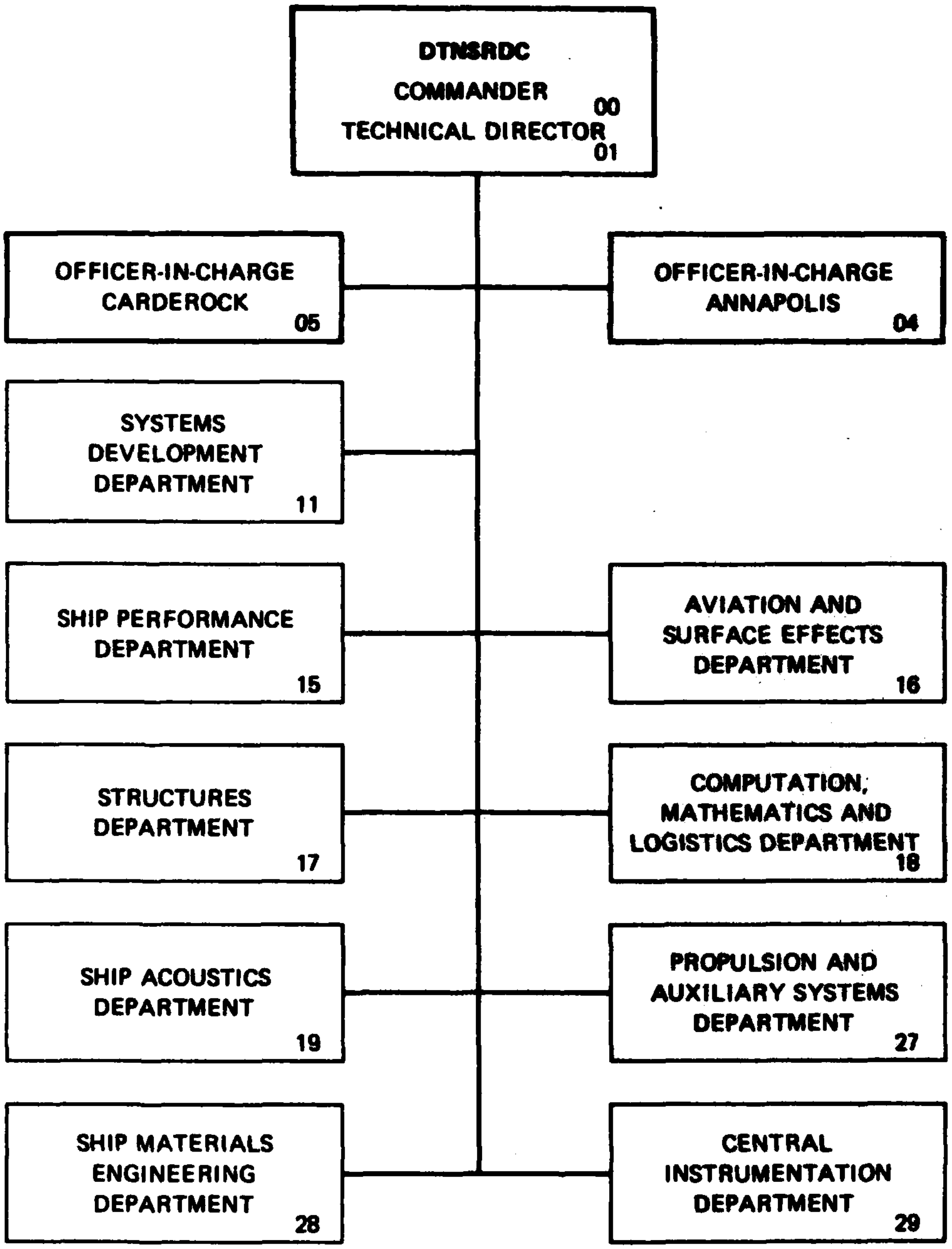
AD-A147 964 FUNDAMENTALS OF COMPUTER PROGRAMMING FOR ENGINEERS(U) 1/2
DAVID W TAYLOR NAVAL SHIP RESEARCH AND DEVELOPMENT
CENTER BETHESDA MD P N ROTH OCT 84 DTNSRDC-84/062
UNCLASSIFIED F/G 5/9 NL



MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

MAJOR DTNSRDC ORGANIZATIONAL COMPONENTS


DTNSRDC
COMMANDER 00
TECHNICAL DIRECTOR 01
OFFICER-IN-CHARGE CARDEROCK 05
OFFICER-IN-CHARGE ANNAPOLIS 04
SYSTEMS DEVELOPMENT DEPARTMENT 11
SHIP PERFORMANCE DEPARTMENT 15
AVIATION AND SURFACE EFFECTS DEPARTMENT 16
STRUCTURES DEPARTMENT 17
COMPUTATION, MATHEMATICS AND LOGISTICS DEPARTMENT 18
SHIP ACOUSTICS DEPARTMENT 19
PROPULSION AND AUXILIARY SYSTEMS DEPARTMENT 27
SHIP MATERIALS ENGINEERING DEPARTMENT 28
CENTRAL INSTRUMENTATION DEPARTMENT 29

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER DTNSRDC-84/062	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) FUNDAMENTALS OF COMPUTER PROGRAMMING FOR ENGINEERS		5. TYPE OF REPORT & PERIOD COVERED Final 6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) Peter N. Roth		8. CONTRACT OR GRANT NUMBER(s)
9. PERFORMING ORGANIZATION NAME AND ADDRESS David W. Taylor Naval Ship Research and Development Center Bethesda, Maryland 20084		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS		12. REPORT DATE October 1984 13. NUMBER OF PAGES 119
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office)		15. SECURITY CLASS. (of this report) UNCLASSIFIED 15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

APPROVED FOR PUBLIC RELEASE: DISTRIBUTION UNLIMITED

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Programming, fundamentals, computer, ratfor, FORTRAN, Pascal, C, basic, debugging

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

This document teaches engineers how to write computer programs independent of any specific programming language.

It shows the four principles of programming and how to use them to build computer programs in English. It shows how to translate the English programs into an appropriate engineering computer programming language: ratfor, Pascal, C, FORTRAN77, FORTRAN66, or BASIC.

It also provides a programming checklist and a debugging guide.

TABLE OF CONTENTS

ABSTRACT

This document teaches engineers how to write computer programs independent of any specific programming language.

It shows the four principles of programming and how to use them to build computer programs in English. It shows how to translate the English programs into an appropriate engineering computer programming language: ratfor, Pascal, C, FORTRAN77, FORTRAN66, or BASIC.

It also provides a programming checklist and a debugging guide.

ADMINISTRATIVE INFORMATION

This document was prepared using Structures Department overhead funds.

INTRODUCTION

I presume you to be an engineer who wishes to write programs to solve engineering problems. You may already have had some experience with computers, but you don't necessarily wish to become a `computer expert.' Unfortunately, use of modern computers requires some dexterity in at least four areas:

- you should have some acquaintance with your computer's `operating system' (the program that lets other programs run);

- you must be able to wield a text editor (the program that lets you type programs and data into your computer);

- you have to be expert enough in your technical field to know that you need to do some computing;

- you have to express the solution to your technical problem in some programming language.

Operating systems and text editors have been addressed in other documents (see [Roth83]* and [Roth82]), and I can't begin to teach you your field, but perhaps I can help you with the programming process itself.

Programs can be written by applying a few simple principles. This text teaches these simple principles in a language you already know, i.e., English. I will further presume, however, that you are trying to become familiar with the syntax of one of the programming languages illustrated here.

Over 85% of the programs in the world are less than 200 lines long, so you will probably spend most of your programming time writing programs of this size. Development of larger programs is a different story because you need a well-

* References are indicated by the first four letters of the author's name followed by the year of publication, in brackets. A complete listing of references is given on page 111.

developed strategy to manage the effort, in addition to knowing the principles of programming. In this slim text, however, we will concentrate primarily on the basic programming principles (which are applicable to programs of any size) and postpone discussion of the management of large program development to a later volume. (Certainly, you must be able to write a small program before you can write a big one.)

The text is organized as follows: The Four Principles of program construction are demonstrated in Chapter 1. Chapter 2 discusses the passive parts of computer programs: data. Chapter 3 describes the Three Parts of every program. Chapter 4, which is fairly long, describes The Constructs (from Control structures) with exercises to test your learning as you go. Chapter 5 describes the process of getting your program to run, and Chapter 6 gives a checklist to help ensure that your program is in the best possible condition before (and after) you run it. Chapter 7 is a short lesson in correcting your program when it doesn't run correctly (yes, even with all of the `good stuff' in the early chapters, things still go wrong). Chapter 8 contains capsule reviews of several programming languages and reveals my not-very-well-hidden bias for certain languages. Chapters 9 through 14 are each devoted to a specific programming language; there is not enough detail presented here to make full use of any language, but enough to get you going successfully. Following the list of references, I have included a small glossary of those terms having special jargon value to programmers.* I therefore recommend that you read Chapters 1 through 6, skim 7 and 8, and select the programming language chapter(s) to read per your own taste.

I have a few other objectives for writing this text. First: algorithms are often published in a language bearing great resemblance to Pascal and Algol. It is desirable to be able to read these algorithms, and perhaps convert them into other languages. By presenting the principles of programming here in several languages, I hope to make it easier for you to read and understand these algorithms.

Second: (a confession) I am comfortable using several kinds of large mainframe computers because I am familiar with the way they behave. I know how to use several text editors and many other software tools, having invented several of my own.

I noticed that I was extremely reluctant to program the Texas Instruments TI59 programmable pocket calculator. I examined my motives for putting off working with the machine, and arrived at the following list of technical and emotional issues:

- The user manual is thick. (`You mean I have to read all this to write a crummy program?')

- There aren't many experts around for me to talk to when I run into trouble, or, they would be busy with their own work (which I would be disturbing); I was `on my own.'

* If you don't understand something on first reading, skip over it. Later text should make it clear.

- I didn´t know what response the machines would make to my commands; consequently I didn´t know what questions to ask when something `went wrong.´

- My program wouldn´t work because I overlooked something very simple; I would therefore appear to be stupid, and be laughed at.

- Work with the machines is tedious. There are <u>no</u> software tools to help program these devices.

- I don´t have time to learn how to use the machine. And running speeds are very slow.

- I don´t believe programmable pocket calculators are effective, so I did not want to become an expert in programming them.

You may have similar apprehensions about using mainframe computers. I believe my excuses (the ones with emotional content) stem from the `fear of the unknown.´ So, in addition to the reasons mentioned above, I wrote this to remove as many unknowns as possible, and thereby reduce the fear level.

Finally: remember that if you can afford to write a couple of drafts of a report or letter, you should allow yourself the `luxury´ of doing two drafts of a program. Be prepared to throw the first draft away; you will anyway. Relax and enjoy yourself.

1. THE FOUR PROGRAMMING PRINCIPLES

Before you write any program, you should ask yourself three questions:

- What is this program supposed to do?

- Has someone else already written one to do this?

- Can several existing programs be hooked together to produce the same effect?

You should be able to answer the first question in one sentence. If you can do what you want to do faster with a pencil and paper, you probably don't need a program.

If the answer to the second or third question is `yes,' you can often save yourself a lot of work by using an existing program instead of writing a new one (you can also skip reading this text). Assuming that a program doesn't exist to do what you want, or an existing one doesn't do it the way you want it done, and there's no one around who can write it for you, you may decide to compose your own. Now you need to know something about programming.*

But you don't need to know a lot! In fact, you need only the following Four Principles:

- Programs `operate on' data of specific types (Chapter 2).

- Programs have at most three Parts (Chapter 3).

- Programs are built from at most six Constructs (Chapter 4).

- Half of the constructs may be altered in at most two ways (Chapter 4).

When you apply these principles, you can compose your program in English (or the reasoning language in which you do your other thinking) and then translate the `English' program into the programming language of choice. The closer the target language is to the way you think, the easier the translation will be. Often, the choice of programming language is predetermined, because you are modifying a program which already exists, or you are writing the program for someone else.

My own practice is to think with a pencil and paper,** making notes in a mixture of English, ratfor, and Pascal, which I then translate completely into the target programming language. Although some panache is demonstrated by developing programs on scraps of paper, envelope rears, and irregular corners of slates, it is advisable to keep the notes you make when you develop your program. These

* Notice that the initial effort in writing a program is to avoid writing a program!

** Rather than erase errors, I find it preferable to cross them out and rewrite correctly. The crossed-out stuff provides a trace on the path of my thinking, which is sometimes helpful to have when debugging time comes.

will help you when your boss insists that you document your work in a more appropriate form, such as a report. It is easier, of course, to write the users´ manual for your program before you write the program itself.

`Efficient´ programs use little computer time and little computer memory. But: computer time and memory are in most cases just too cheap to worry about! Your time is infinitely more valuable than silicon time. Thus, efficiency is that which makes your life `easier.´ Paraphrasing Boudreau [Boud71], "A big, slow, correct program is better than a slick wrong one." So get it right before you make it `fast.´ As you gain experience using the four Principles, you will notice that you are producing programs which are not only correct, but readable, understandable, and quite fast, as well.

The action "translating completely into the target programming language" is called "coding." Because translating incomplete thoughts into a programming language is quite difficult, please regard the following maxim:

THE SOONER YOU START CODING,
THE LONGER IT WILL TAKE TO FINISH YOUR PROGRAM.

When the translation from English to programming language is complete, the program can be typed into a computer (see [Roth82] for help in this area), and compiled and executed using the procedures of [Roth83]. Each of the language chapters in this report also shows how to do it.

2. DATA

2.1 Data, Constants, and Variables

Data are the objects on which computer programs operate.

Constants are items of data which do not change.

Variables are things which may assume the values of different constants.

2.2 Types

We can assign values to, and examine the values of, variables according to the type of the data.

The data types useful for the majority of straightforward engineering programs are:

integer — Numbers which have no fractional part. Most often used to count things. For example, the number of keys on a piano is an integer constant (88). A variable used to store this data type is called an `integer variable.´

real — Numbers which may have a fractional part. Useful for representation of physical quantities like mass, density, thickness, location in space, etc. The ratio of the circumference to the diameter of a circle is a real constant: 3.14159.... A variable used to store this data type is called a `real variable.´

Boolean — (or logical). An item of Boolean data may have only one of two possible values: `true´ or `false.´ This tvpe allows control of programs through logic. It is `true´ that 2.0 is greater than 1.0. A variable used to store a Boolean constant is called a `Boolean variable.´

character — Used to read and write the character set available on the machine. These are typically the lower case letters, the upper case letters, the digits, the space, the tab, special symbols such as !, @, #, etc. A variable used to store this data type is called a `character variable.´

2.3 Expressions

Each of the types is defined for an expression in a language. A valid expression is a combination of constants, variables, and symbols which conforms to the syntax of the language. You can assign the value of an expression to a variable of the appropriate type via the assignment statement of the language (more on this later).

2.4 Data Organizations

The data organizations most useful for straightforward engineering programs are the simple variable, the array, and the file.

2.4.1 The Simple Variable A simple variable is a thing to which you can assign values according to its type, compare its value with variables or expressions of the same type, and write the value of the variable (so you can see the results of assignments and comparisons).

All computing with variables is done in accordance with the variable´s type. For example, we can assign "position on the x-axis" to a variable we name X of type real. Or we can assign the truth of the expression "X is greater than 6.1" to a variable B of type Boolean.

2.4.2 The Array An array is an ordered, determinate number of variables. You should think of an array as a single entity, even though it may have many parts. For example, the direction cosines of a vector form a single array of three real data items. We might name this array DIRCOS.

The generalization from simple variable to array suggests `an array of arrays,´ `an array of arrays of arrays,´ etc. For example, we can consider the location of a point in space to be given by the three real coordinates of the point: X, Y, and Z (the names of three real simple variables). Or, we can consider the location to be given by an array which contains three real variables. Suppose we name this array `X.´ Then X(1) could be the `x´ coordinate, X(2) the `y´ coordinate, and X(3) the `z´ coordinate.

Further, if we have 10 points in space, we can conceive of them as an array of arrays. Suppose we name the array COORD. Then we need 10 `places´ in the array (one for each point). In each of the `places´ we need an arrav of three real variables (one for each axis). So,

COORD(1,1) might be the `x´ coordinate of the first point,

COORD(4,2) could be the `y´ coordinate of the fourth point,

COORD(7,3) could be the `z´ coordinate of the seventh point.

In the COORD array, therefore, the first `subscript´ gives the number of the point and the second gives the axis of measurement. The important thing to note is that the array concept allows us to deal with a lot of numbers with a single name.

Note that although each item of an array can be of any type (the example above used an array of type real), the subscripts are always of type integer. By using integers in control structures, we can process arrays very quickly and with great economy of language.

2.4.3 The File A file is an ordered collection of variables which may be of unknown length. To indicate a file, I will use a name surrounded by brokets: <input> is to be read `the file whose name is input,´ or simply `the input file.´ Examples:

- <input> to a finite element program is an unknown number of `lines,´ each line being an array of 80 variables of type character.

- This report originated as several files on a UNIX* system (at the time I first wrote this sentence, the files were 2306 lines long, and had 9020 `words´ made up of 51560 characters).

Files are used for input and output to computer programs. Although the `memory´ of a computer is indeed finite, the files available to a program are conceptually infinite. This allows us a handy place to temporarily save large numbers of simple variables and arrays which would exceed the capacity of the computer´s memory.

Because of the possibility that files can be very long, programming languages usually provide access to files through a "window;" that is, you only get to see a little of it at a time. The shape of this window is greatly dependent on the operating system and the language being used.

2.4.4 Other types and organizations Although the types and organizations mentioned above are satisfactory for the vast majority of the programs which you will write, the list is incomplete. FORTRAN, for example, provides the additional types

complex A variable which has a `real´ and `imaginary´ part; both of the parts are represented internally as `reals´.

double A variable which has twice the precision of a `real.´

Other languages provide other types and organizations which include the SET, the RECORD, the TREE, etc. To use these in a language which does not explicitly provide them requires ingenuity in mapping the structures. You are limited only by the limits of your imagination, and the capability to express your idea in terms of a known language running on an available machine.

The best treatment of data types and organizations is given by [Wirt76], which is based on the Pascal language. In FORTRAN, [Day72] and [Berz71] are quite good.

* UNIX is a trademark of Bell Laboratories.

2.5 Variable Names

> "Don't stand there chattering to yourself like that," Humpty Dumpty said, looking at her for the first time, "but tell me your name and your business."
>
> "My name is Alice, but--"
>
> "It's a stupid name enough!" Humpty Dumpty interrupted impatiently. "What does it mean?"
>
> "Must a name mean something?" Alice asked doubtfully.
>
> "Of course it must," Humpty Dumpty said with a short laugh: "my name means the shape I am - and a good handsome shape it is, too. With a name like yours, you might be any shape, almost." [Carr96]

--- + ---

Taking the Humpty Dumpty hint, we will wish to name variables in such a way that their names give us a clue to their essence. Because English is our language, we have no limitations on the length or style of a variable name. What is interesting about English (or mathematical) variable names is that they are assigned in a context which preserves their meaning. Thus, when we say something like, `Let P be a point in space,' then the name `P' somehow is seen to be tied to the word `point.' Of course, a second point would need another name to make it distinct from P. Mathematicians will usually try to pick a name that is somehow close to the first name, and yet obviously different; perhaps `Q' (it is `near' P alphabetically). We note that P and Q are short names because we usually prefer not to write very much. At the same time, names (i.e., words) which are longer than eight characters are usually at least three syllables, which makes them hard to read and harder to type.

On the other hand: a textbook which defines `P' on page 2, say, and never reminds us that P is a point, is going to be a difficult text to refer to. That's why mathematicians are always redefining things for their readers; context is usually limited to about a page or two. You should cultivate the same habit in your programs, because programs are not always read from beginning to end; one enters them at any place suitable to the need at hand. It is therefore important that the program reader be able to get a fair understanding of what is going on from the immediate context.

Thus:

VARIABLE NAMES SHOULD
BE OF `REASONABLE' LENGTH.

`Reasonable' means from one to eight characters in length (maximum length of twelve characters), or redefined with commentary at fairly frequent intervals in the program text. In the `Let P be a point in space' case, my practice would be to name the variable P in a program of less than 50 lines, and name it POINT in any program of 50 lines or larger.

The creative act of naming variables is difficult! A thesaurus can be quite helpful.

Exercise: Obtain someone's program (preferably one of your own that is at least 6 months old), and take a look at the variable names. Can you tell what they are? Why or why not?

3. THE THREE PARTS OF EVERY PROGRAM

Computer programs have three parts:

- The Initialization.
- The Computation.
- The Clean-up.

3.1 The Initialization

This program part is that in which we get ready to compute. It includes

the program header line,

the declaration of variable data types,

the opening and positioning of files,

the setting of default values, and

the setting of initial conditions.

On some computers, the operating system itself provides much of the program initialization. The pieces most often under your control are the declaration of variable type, and the initialization of variables to some value. The declarations are much like the statements preceding a mathematical proof ("Let f be a smooth continuous function in the domain D with f′ the first, and f′′ the second, derivative of f with respect to distance in D").

Initialization of a variable to a specific value uses the `assignment statement,′ which `assigns′ a value to a variable. Because the form of the assignment statement varies from language to language, each language chapter includes a section defining this statement.

The initialization is usually the last part of a program to be completed.* This is because you don′t know what the initial values should be when you begin composition of the program. Hence, initialization can also be the hardest part of a program to write.

* This is comparable to the way in which a report or book is written: the last parts to be written are the introduction and the abstract.

3.2 The Computation

This is the program part which does your work. It is the part where you will use The Constructs.

3.3 The Clean-up

In this part of the program, we save those things which we need to keep, throw away the garbage, and exit from our computations gracefully (if possible).

On many computers, the entire clean-up is performed by the operating system.

4. THE CONSTRUCTS

In this fairly long chapter, I present The Constructs both in English and graphically.

4.1 Boclean Logic

The facility for logic built into programming languages is a mechanical implementation of Boolean logic. Boolean expressions are those which allow us to test truth and falsity according to the rules of logic formalized by Boole. A Boolean expression may have only one of two values: TRUE or FALSE.

4.2 Form of Boolean Expressions

In the following, X and Y are variables or expressions of the same type, and each `Boolean expression´ will be either TRUE or FALSE.

X is greater than Y.

X is greater than or equal to Y.

X is equal to Y.

X is not equal to Y.

X is less than Y.

X is less than or equal to Y.

4.3 The Assignment Statement

The actual activity of a program is performed by statements which conform to the language syntax. A statement in a language will be represented here in the form `S´, `S1´, `S2´, etc.

One of the most important statement types is the assignment statement, which has the form

X ao E

in which `X´ is a variable, `ao´ is the assignment operator (usually a symbol composed of 1 or more characters), and `E´ is an expression of the same type as `X´. One reads an assignment statement: "the current value of `X´ is replaced with the value `E´" or, more simply, "`X´ becomes `E´." This is obviously different from the statement "`X´ equals `E´." The latter is a `Boolean expression´ which may assume only the value TRUE or FALSE; the former is called the `assignment statement´ because the value of an expression is assigned to a variable.* Note: execution of the assignment statement destroys any previous value of X!

We emphasize that the assignment statement is not the same as the Boolean expression `X equals E´; the assignment statement is an action, whereas the Boolean expression is an assertion.

For example, let´s assume that the integer variable X has the value 1. Then

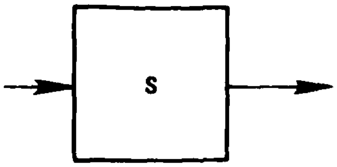
X equals X + 1

is false, whereas

X becomes X + 1

assigns the value 2 to X.

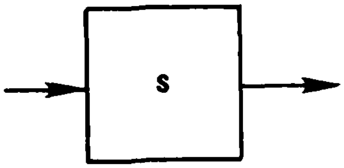
A statement will be diagramed



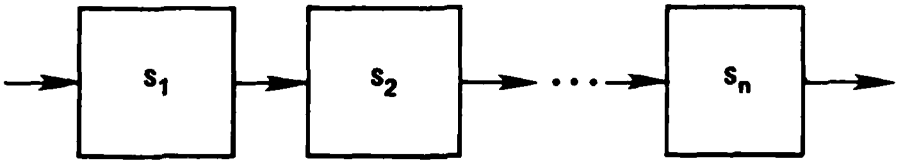
where the arrow into the box indicates that control is passing to the statement `S.´ When the statement `S´ has completed its action, control passes to the `next´ statement to be executed; this is represented by the arrow out of the box.

* The expression may be as simple as another variable, in which case we say that one variable is copied into another.

In general, the notation `S´ may be read as `n statements in sequence´ where "n" is greater than or equal to 0. Thus,



may actually represent something like



Most contemporary computers are sequential machines, which means that statements are executed in the order in which they are accessed. Thus, if we write the statements S1, followed by S2, followed by S3, the statements will be executed in the order S1, S2, S3 unless we somehow change this order. This `changing´ is done with The Constructs. Please note that each of The Constructs is also a `statement,´ and may be hooked together just as any of the `other´ statements of a language are.

4.4 The Conditional Construct

The Conditional construct permits computations to be performed only if certain conditions are met. Given that B is a Boolean expression, the Conditional construct has the form

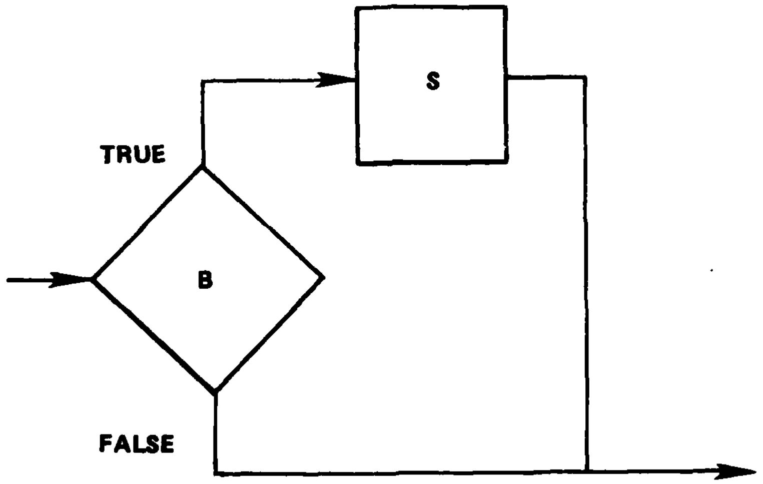
```
If B,
   then S.
```

For example:

```
If "it is raining"
   then "carry an umbrella."
```

In this example, the Boolean expression B is: "it is raining." This expression may be TRUE or FALSE. The action to be performed, S, is "carry an umbrella."

The Conditional construct may be diagramed



where the `if´ part of the statement is represented by the B inside the diamond. If `B´ is true, then the true path is taken, and `S´ is done. If `B´ is false, then the false path is taken.

4.5 The Alternative Construct

When there are alternative actions which may be performed, we can use the Alternative construct. Given that B is a Boolean expression, then the Alternative construct has the form

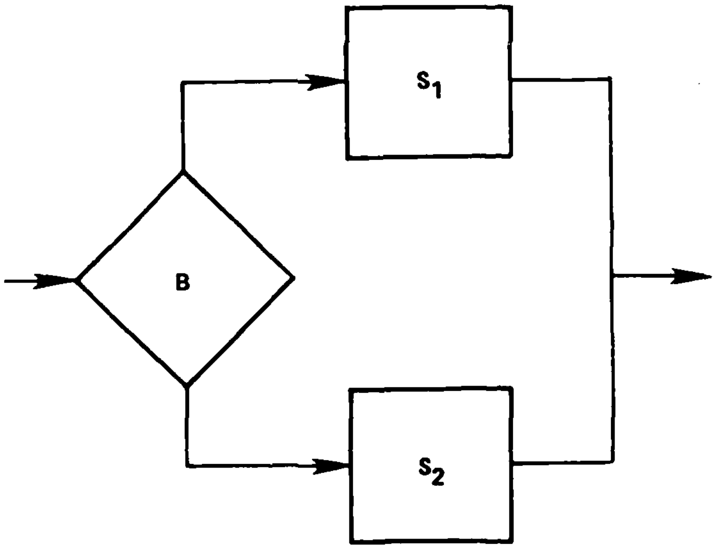
```
If B
   then S1;
else,
   S2.
```

For example:

```
If "it is raining"
   then "carry an umbrella;"
else,
   "lower convertible top."
```

Note that these actions are mutually exclusive. That is, "if it is raining" then the only statement which is executed is "carry an umbrella." "If it is not raining," then the only statement which is executed is "lower convertible top."

The Alternative construct can be diagramed



4.6 The Multiple Choice Construct

Out of a number of possibilities, we may wish to select the appropriate one and do the statement(s) associated therewith. For example:

```
In case of
   rain: "carry an umbrella"
   snow: "wear parka and hat"
   hail: "stay home"
   tornado: "goto cellar"
   sunshine: "play tennis"
```

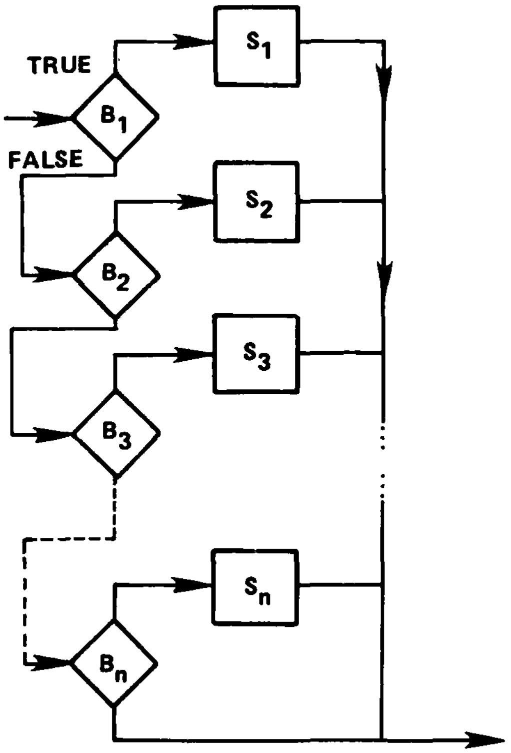
where only one of the choices is elected. Thus, "goto cellar" and "play tennis" are mutually exclusive options.

Another way of expressing this construct is with a succession of Alternative constructs:

```
if "it is raining" then
   "carry an umbrella"
else if "it is snowing" then
   "wear parka and hat"
else if "it is hailing" then
   "stay home"
else if "it is tornadoing" then
   "goto cellar"
else if "it is sunshining" then
   "play tennis"
else
   "turn on radio"
```

where we have included the last `else´ to take care of the time when none of the weather conditions we know about occurs.

The Multiple Choice construct can be diagramed



4.7 Iteration

Iteration is one of the great strengths of computers: the machines will do the same actions over and over with no complaint.

There are two types of iteration: indeterminate, and determinate. Indeterminate iteration can be profitably applied to reading numbers from files when the length of the file is unknown prior to reading. Determinate iteration may be used when certain portions of an array whose size is completely known must be set to some value.

Indeterminate iteration may be classified into two subtypes:

- Test, then (perhaps) act. This iteration construct is called the While.
- Act, then test. This construct is called the Repeat-Until.

4.7.1 The While Construct The indeterminate iteration is applicable in those cases when the number of times the iteration will be done is not known a priori. Given that B is a Boolean expression, then the While construct has the form

```
while B,
   S.
```

Note that the statement S may not be executed at all! For example:

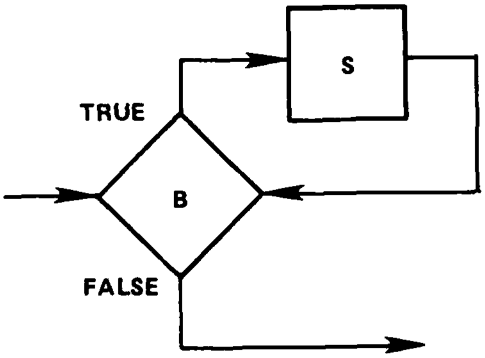
```
while "not quitting time"
   "do some more work"
```

If it is quitting time, then "do some more work" is not done. Another example:

```
While "you have not reached the end of a file"
   "read and process the next line of the file"
```

If we are already at the end of file, then "read and process the next line of the file" is not even attempted.

The While construct can be diagramed



The `going back´-ness of this construct is what gives rise to the term `loop´ to describe iteration. I will henceforth use the terms `iteration construct´ and `loop´ interchangeably.

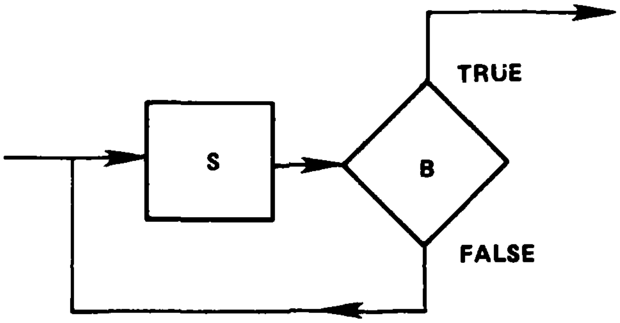
4.7.2 The Repeat-Until Construct This iteration places the test at the end of the loop. Note the difference between the While and the Repeat-Until: the statements controlled by the While may not be executed, whereas the statements under control of the Repeat-Until are guaranteed to be executed at least once. Given that B is a Boolean expression, then the Repeat-Until construct has the form

```
Repeat
   S1.
   S2.
Until B.
```

In this construct, the statements S1 and S2 are guaranteed to be executed at least once. For example:

```
Repeat
   "brush uppers"
   "brush lowers"
Until "teeth are clean"
```

The Repeat-Until can be diagramed



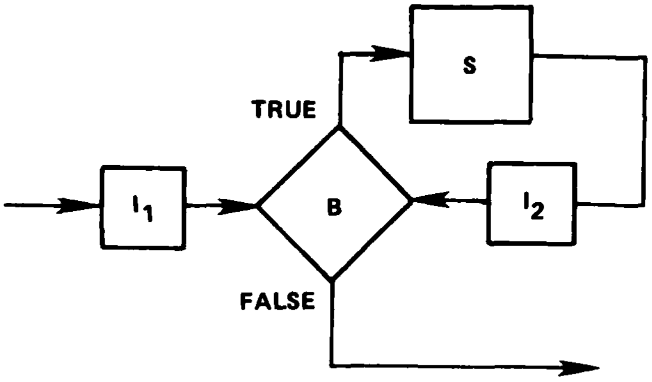
4.7.3 <u>The Determinate Iteration Construct</u> This kind of iteration is called determinate because the number of times the loop will be executed is dependent on an index to some countable sequence. The .sequence is assumed to begin at J and continue to K in increments of M (the sequence may increase or decrease).

Let B be a Boolean expression which compares the values of the integer variables I and K (say, for example, B is the expression "I is less than K"). Then the determinate iteration construct has the form

```
I becomes J.
While B {
   S.
   I becomes I + M.
   }
```

where the braces `{´ and `}´ define the complete range of statements under control of the While. Note that there exists the possibility that statements "S" and "I becomes I + M" may <u>not</u> be executed at all (why?).

Determinate iteration may be diagramed



where the statement I1 represents the initialization of the counter, and I2 represents the resetting of the counter so it points to the next item of interest.

As an example of this construct, let´s find and print the largest element MAX in an array of integers ZOT which has N elements. Let´s say N = 3 and the contents of ZOT are

ZOT(1) = 27
ZOT(2) = 31
ZOT(3) = 12

Let´s assume we have an integer variable `I´ which we can use to access each of the elements of the array ZOT. We´ll also assume that the contents of ZOT are calculated in some other part of our program, and that we don´t know what the values of ZOT are. We <u>do</u> know that there are N values we have to look at, and that we´d better look at them all. So, here is an English `code fragment´ which will allow us to find MAX

```
MAX becomes ZOT(1).
I becomes 1.
While "I is less than or equal to N" {
   If "MAX is less than ZOT(I)" then "MAX becomes ZOT(I)"
   I becomes I+1.
   }
Print: "The largest value in the ZOT array is " MAX.
```

In this example, we´ve combined the While and the Conditional constructs.

Exercise: mark the three parts of this program fragment to show the initialization, computation, and clean-up phases.

Now let´s step through the calculations by substituting numerical values for each variable:

```
MAX becomes 27.
I becomes 1.
While "1 is less than or equal to 3" {
   If "27 is less than 27" then (not true, so do nothing!)
   I becomes 2
While "2 is less than or equal to 3" {
   If "27 is less than 31" then "MAX becomes 31."
   I becomes 3
While "3 is less than or equal to 3" {
   If "31 is less than 12" then (not true, do nothing)
   I becomes 4
While "4 is less than or equal to 3" (not true, end the While)
The largest value in the ZOT array is 31. (printed answer)
```

and the loop terminates with MAX = 31 and I = 4.

Exercises:

1. Initialize I to 2 instead of 1. Is the program any faster? By how much? Is it easier or harder to understand? Why?

2. Does the changed program work if N = 1?

3. How would you change this program to make it work for N = 0? What value would you assign to MAX?

4.8 Altered Loops

Strictly speaking, we can structure our code. with only the Conditional, the Alternative, the Multiple Choice, the While, the Repeat-Until, and the Determinate Iteration constructs. Using only these constructs sometimes makes for extremely convoluted programs, and the meaning of a program can actually be made clearer by performing a test inside a loop and directing the path of the execution according to the test results. This kind of construct thus alters the nature of the loop to `semi-determinate.´ Be this as it may, please observe the following maxim:

ALTER LOOPS INFREQUENTLY.

You will notice in the graphics for altered loops that there are seven paths between parts of the construct. Although this is still within the realm of comprehension, it is complicated, and makes understanding of a program more difficult.

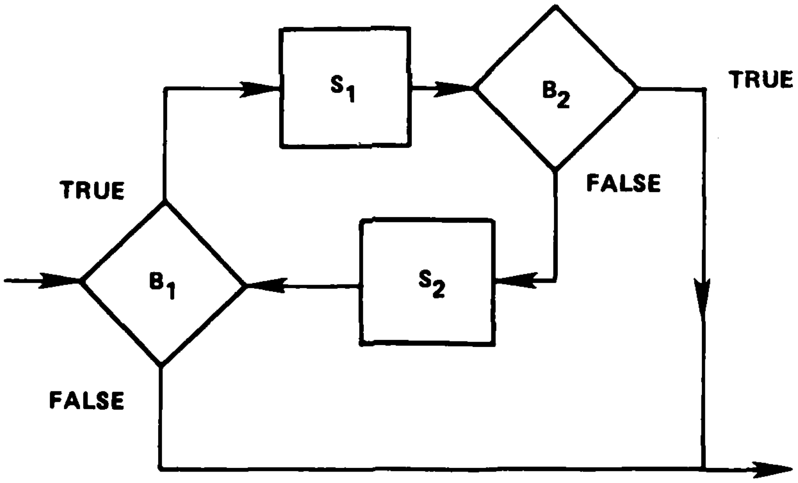
4.8.1 Loop Exits

The first unusual circumstance is `stop iterating immediately (i.e., leave the loop);' the next statement to be executed is the one which follows the terminator of the loop. This construction is typical in parts of programs which check data for errors and reach an impasse.

```
   While B1 {
      S1.
      If B2
         then "break"
      Else,
         S2.
      }
   S3.
```

In this construct, the word "break" means that computation is to continue with statement S3, which is entirely outside of the range of the While.

The Loop Exit can be used to alter any iteration construct; it is diagramed modifying a While.



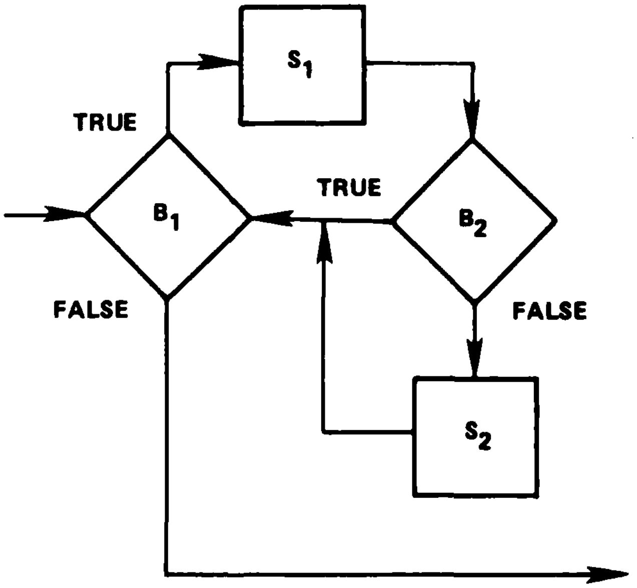
4.8.2 Loop Redo

The second unusual circumstance is `immediately continue iteration with the next test.´

```
While B1 {
   S1.
   If B2
      then "next"
   Else,
      S2.
   }
S3.
```

In this construct, the word "next" means that computation is to proceed with the next test B1. That is, "next" says "continue computation with the statement `While B1´." By the way, use of the "next" is extremely rare (I have used it infrequently, and find few uses in the references).

The Loop Redo can be used to alter any iteration construct; it is diagramed modifying a While.



4.9 Construct Summary

```
Conditional:        If B then
                       S.

Alternative:        If B1 then
                       S1.
                    Else
                       S2.

Multiple Choice:    In case of                          If B1 then
                      B1: S1.                              S1.
                      B2: S2.          (or)             Else if B2 then
                      B3: S3.                              S2.
                      B4: S4.                           Else if B3 then
                                                           S3.
                                                        Else if B4 then
                                                           S4.
While:              While B
                       S.

Repeat-Until:       Repeat
                       S
                    Until B.

Determinate:        I becomes J.
                    While B {
                       S.
                       I becomes I + M.
                       }

Loop exit:          Break.

Loop Redo:          Next.
```

4.10 Which Construct?

We now have a complete set of constructs to apply to any programming problem. The nagging question remains: Which construct should be used where? You will usually have little difficulty in selecting from the Conditional, Alternative, or Multiple Choice constructs. The problem usually lies with the loops.

The following may help in selecting the appropriate iteration construct:

- How the iteration is supposed to stop will hint at the form of the construct. By the way, make sure it will stop!

- You can always use a While (you might have noticed that the two other iteration constructs are really variations of the While).

- If you can count the data items to be processed, then the Determinate iteration is usually proper.

4.11 Examples

These examples are a good opportunity for some exercise. For each case, write an `English´ program, and compare it to the sample solution shown at the end of the chapter. Then translate your program into a programming language and compare your translation with the example shown in the section devoted to that target language.

4.11.1 Temperature Conversion Produce a table of equivalent Celsius temperatures for the Fahrenheit temperatures from -40F to 100F in increments of 5F.

Given any Fahrenheit temperature F (i.e., F is the REAL temperature), the Celsius, or PHONY, temperature C is given by the equation

```
C = 5*(F - 32)/9
```

where the symbol `*´ means `multiply,´ and the symbol `/´ means `divide.´

We might proceed as follows. Beginning at the top of a page of lined paper, write down the first Fahrenheit temperature to be converted: -40. We now plug -40 into our recipe for Celsius temperature

```
5*(-40-32)/9 = 5*(-72)/9 = -360/9 = -40
```

and write down the answer -40C in a second column. Since we haven´t completed our job yet (we haven´t reached 100F), we add 5F to the current temperature (-40F) to obtain -35F. With Fahrenheit temperature of -35F, we plug and chug, producing -31C. We continue this process until the table is full.

Space is provided here for your program.

4.11.2 Nearest Points Given 10 points in 3-D space, find the two points which are nearest to each other.

The solution to this problem is intuitively obvious. We merely compute the distance from each point to its neighbors using the distance formula

```
   d = sqrt[(x2-x1)**2 + (y2-y1)**2 + (z2-z1)**2]
```

where `d´ is the distance, `sqrt´ represents the square root, `**2´ represents squaring a number, `x2´ is the x-coordinate of "one" point (y2 and z2 are the y- and z-coordinates of this same point), `x1´ is the x-coordinate of "another" point (y1 and z1 are the y- and z-coordinates of this same point).

Those two points which are at the minimum distance are the two closest neighbors.

Additional exercises:

1. Modify your program to find the points which are farthest apart.

2. Print all pairs of points which are at the same minimum (maximum) distance.

3. Print a table of the distances from each point to its neighbor.

4.11.3 Count the `A´s Read <input> and count the number of `A´s in the file.

This is a simple problem mentally, but is quite tedious and prone to error. Let´s assume that we can tell when we have reached the end of a file, much like we can tell when we´ve reached the end of a book. Then we need merely read the file a character at a time, and check to see if the character we´ve just read is an `A.´ If it is, then we can increase our count of `A´s by one. (Hint: What does the initial value of our count have to be, i.e., before we read any characters? Does your program give the correct count when there are no `A´s? When there are no characters at all?)

Additional exercise: modify your program to read real numbers from a file and count the values greater than some threshold, say, 10.0. Assume that the last line in the file will have the number -99999.0.

4.12 Solutions to the Examples

4.12.1 Temperature Conversion

Initialization - declarations:

Let F and C be real variables which measure temperature.

Initialization - setting initial values:

F becomes -40.

Computation:

```
While F is less than or equal to 100 {
  C becomes 5*(F-32)/9.
  Print: F, C.
  F becomes F + 5.
  }
```

Cleanup:

(none!)

4.12.2 Nearest Points

Initialization - declarations:

```
Let OLDD be a real variable measuring the smallest distance.
Let NEWD be a real variable measuring
   the distance between any two points.
Let X be an array of reals containing the 3 coordinates of
   each of the 10 points. Let the first subscript designate the point,
   and the second designate the coordinate.  X is therefore 10 by 3.
Let I, J, K, M, and N be integer variables used for counting.
```

Initialization - setting initial values:

```
Read all the coordinates into X from <input>.
OLDD becomes the distance between point 1 and point 2 (why?);
   i.e., OLDD becomes SQRT((X(1,1)-X(2,1))**2
                         + (X(1,2)-X(2,2))**2
                         + (X(1,3)-X(2,3))**2)
```

Computation:

```
I becomes 1.
While "I is less than or equal to 9" {

  J becomes I+1.
  While "J is less than or equal to 10" {

    NEWD becomes SQRT((X(I,1)-X(J,1))**2
                    + (X(I,2)-X(J,2))**2
                    + (X(I,3)-X(J,3))**2)

                                          (found a new nearest?)
    If "NEWD is less than or equal to OLDD" then {
      M becomes I.
      N becomes J.
      OLDD becomes NEWD.
      }

    J becomes J+1.
    } (end of While "J is less...)

  I becomes I+1.
  } (end of While "I is less...)
```

Cleanup:

```
Print: ´ Closest points are:´
Print: X(M,1),X(M,2),X(M,3)
Print: X(N,1),X(N,2),X(N,3)
```

4.12.3 Count the `A´s

Initialization - declarations:

```
Let Count be an integer variable.
Let Ch be a character variable.
```

Initialization - setting initial values:

```
Count becomes 0.
Read Ch.
```

Computations:

```
While "not end of report" {
  If Ch = `A´ then
    Count becomes Count + 1.
  Read Ch. (destroying what was in Ch)
  }
```

Cleanup:

```
Print: Count.
```

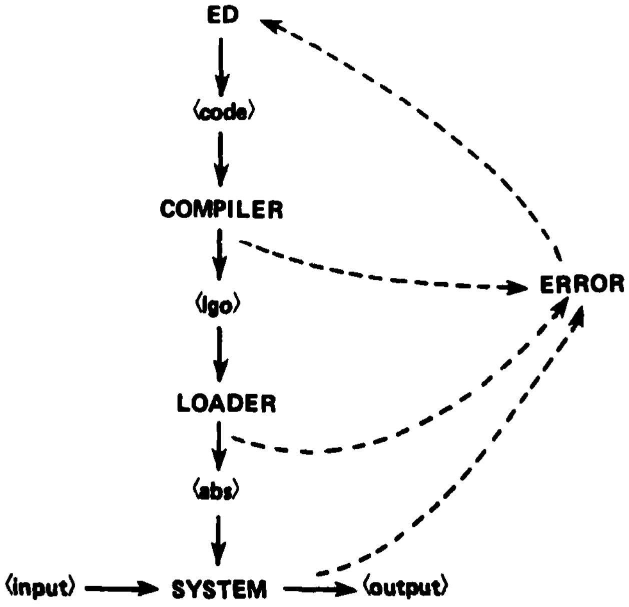
5. GETTING YOUR PROGRAM TO RUN

Assuming that you've written your program correctly in `English' and correctly translated it into a programming language, how do you get it onto the machine and running?

Presuming you are working with a CDC machine, the steps necessary are:

- typing the program into the computer using a text editor (with ED, for example [Roth82]),
- `compiling' the code into `machine language,'
- `loading' the machine language,
- and `executing' the program.

A sketch of this procedure is shown below.



I've called the file of code that you write with ED `<code>.' It will be a file of, say, FORTRAN, ratfor, or Pascal.

A COMPILER is a program which translates a `higher order language' (your code) into `machine language,' which is very difficult to read and understand, and which varies from one brand of machine to another. Compilers were invented to reduce your need to look at the machine language, and express yourself in a way which is portable from one machine to another.

The machine language file is called <lgo> (from load and go). It is difficult (impossible) to read <lgo> with ED; if you do enter ED with a <lgo> file, you will destroy it if you do a SAVE!

GETTING YOUR PROGRAM TO RUN

`Loading´ is a process which allows you to use things which other people have written, including the transcendental functions, the input/output routines, graphics routines, etc. Only files of machine language are LOADed, so the process is largely invisible to a casual programmer. Naturally enough, LOADing is performed by a program called a LOADER, which reads your <lgo> file and writes its product, the `absolute´ program, into a file called <abs>. <abs> is the only kind of file which can execute on a CDC machine (<abs> is also impossible to read with ED).

Execution is the performance of work according to the program you wrote. This is done by typing the command `abs.´

The solid lines in the sketch show the path of intellect when everything `goes right.´ Unfortunately, everything going right on the first shot is a rare occurrence. Still, you want to spend as little time as necessary in this `loop´ trying to get your program to run. You can use the aids in the next two chapters to make everything `go right´ as soon as possible.

The dotted arrows show the path of intellect when something `goes wrong.´ You eliminate errors by repairing <code> and repeating the compile-load-go process until no errors exist, since the tools used here are very reliable. That is, except in extremely rare cases, the errors are yours, not the computer systems´.

The easiest ways to effect this loop quickly are through the use of the procedures in Chapter 4 of [Roth83].

6. THE PROGRAM CHECKLIST

Although programs can be designed and coded from simple principles, the devil hides in the details. To prevent this ancient foe from getting a toehold on their code, experienced programmers can often be observed practicing seemingly weird and lengthy incantations before they run their programs. The list presented in this chapter extends one found in [Myer79] and initiates the novice into this cabala. Scan through this list and compare your program to each of the questions asked here.

6.1 Data Reference Errors

1. Is a variable referenced whose value is unset or uninitialized?

2. Are there any variables which are not referenced? Is this because you misspelled a variable name?

3. Are all array references within the bounds of the array size?

4. Are all array references selected with an integer subscript? Some languages allow REAL type variables as array subscripts (it is best to avoid this `feature´). If you do this, is the subscript what you expect it to be?

5. Is a variable being assigned a value which does not match its type?

6. Are there any `off-by-one´ errors in referencing array elements?

7. If an array is referenced in several procedures or subroutines, is it defined identically in all sections?

6.2 Data Declaration Errors

1. Have all variables been explicitly declared? Undeclared arrays can be misinterpreted by some compilers as functions.

2. Is each variable declared to have the correct type?

3. Are all variables and arrays initialized properly? I.e., do the values assigned to each variable agree with the types of each variable?

4. Are there any variables with similar names (e.g., POINT and POINTS)? This is not necessarily an error, but it is a sign that names may have been confused or misspelled somewhere in the program.

6.3 Computation Errors

1. Are computations done with variables having inconsistent types (e.g., multiplication of variables of type `character´)?

2. Are there any mixed-mode computations (INTEGER and REAL)? This is not necessarily an error, but is the computation result of the expected type?

3. Do computations with arrays have the proper matching lengths where required? E.g., a matrix product requires a match of the `inner dimension′ of the arrays.

4. Are any divisions performed with divisors very close to zero? Does this affect the validity of the computations?

5. Is it possible for a divisor to be zero? Is it possible that some functions employ a zero divisor (e.g., the MOD function, the arctangent function, etc.)?

6. Are there any consequences of the fact that digital computers rarely represent decimal numbers exactly? I.e., 1/3 + 1/3 + 1/3 does not equal 1.0.

7. Can a variable go outside its meaningful range? For example, can a variable measuring probability ever be greater than 1.0?

8. Is the assumption of the order of evaluation correct for expressions which contain more than one operator?

9. Are there any invalid uses of integer arithmetic? For instance, if I is an integer variable, 2*I/2 is equal to I only if I is even and only if the multiplication is done first.

6.4 Comparison Errors

1. Are there any comparisons of variables of incompatible types? I.e., are characters compared with reals?

2. Are there any mixed mode comparisons?

3. Are the comparison operators correct? Most of the difficulty arises in the combined use of `and′, `or′, and `not.′

4. Are the operators of logical expressions of type logical? For example, to determine if I is between the values 2 and 10, the correct expression is (2<I)&(I<10), not (2<I<10).

5. Are there any comparisons of numbers with fractional parts in which truncation errors play a role?

6.5 Construct Errors

1. Is it possible that certain entry conditions will prevent execution of your program? For example, in the loop

```
while "not found"
   S.
```

what happens if "found" is initially true?

2. Can the number of selections in a Multiple Choice construct ever exceed the number of possibilities you've allowed for?

3. Will every loop eventually terminate?

4. What are the consequences of an Indeterminate Iteration going all the way to the bitter end? For example, in the following program fragment (a loop controlled by a compound Boolean expression), what happens if "found" never becomes true?

```
i becomes 1.
while ( (i < tablesize) and (not found)){
   S.
   i becomes i + 1.
   }
```

5. Are there any "off-by-one" errors (too many or too few iterations)?

6. Is there a corresponding ending bracket to every opening bracket (if the programming language uses them)? Are the program statements grouped properly with brackets?

7. Are there any non-exhaustive decisions? E.g., if a variable is supposed to have one of the values 1, 2, or 3, do you assume that the value must be 3 if it is not 1 or 2? Is this valid, particularly for program input?

6.6 Interface Errors

Some programming languages allow programs to be broken into smaller, more manageable parts called subroutines or modules. The interface between modules can sometimes be a source of errors.

1. Is the number of arguments received by a module the same as the number sent? Are they in the correct order?

2. Are there any unused arguments?

3. Do the attributes of each passed variable match the attributes of the received variable? For example, is a simple REAL variable passed to a module which expects an array?

4. Are the units of each passed variable correct? For example, is the passed value expressed in degrees while the expected value is expressed in radians?

5. Does the number of variables passed by a module equal the number of variables expected by the called module? Are they in the correct order?

6. Are the number, order, and type of variable correctly passed to `built-in' functions? For example, does the arctangent function require one or two arguments? If two, which one is the divisor?

7. Does a subroutine alter a value which is supposed to be only input? For example, is an argument defining the length of an array used as the index to a determinate loop? What is it's value on return from the module?

8. Are global variables referenced the same way in all modules? In FORTRAN, for example, are the variables listed in COMMON blocks the same everywhere?

9. Are constants ever passed to subprograms? For example, the FORTRAN statement

   ```
   CALL SUB(A,3)
   ```

   can be dangerous, because if SUB assigns a value to the second argument, 3 will no longer be 3! (This has given rise to the old saw: `All constants are variable.')

10. Will the program, subroutine or module eventually terminate?

6.7 Input/Output Errors

Many languages permit several ways to access files. If you are using these features, check your program with this list:

1. Are file attributes correctly declared?

2. Are the attributes on the OPEN statement correct?

3. Does the format specification agree with the READ statement?

4. Are arrays declared to be large enough to contain all the information to be read?

5. Have all files been opened before use?

6. Are end-of-file conditions detected and handled correctly?

7. Are Input/Output error conditions handled correctly?

8. Are there spelling or grammar errors in any messages written by the program? Are the messages intelligible?

6.8 Miscellany

1. If the compiler you are using has a `post mortem dump' switch, set it to `on.' This switch will allow the computer to print, at the time of your program's death, the values of your variables according to their types (rather than in their octal or hexadecimal representation). I.e., the values of REAL variables are printed in engineering notation, INTEGERs are printed as integers, LOGICALs are printed as TRUE or FALSE, etc. The clues offered by this tool are extremely helpful. (Note that the procedures of Chapter 4 of [Roth83] have this switch set `on'.)

2. Does the compiler cross reference map indicate variables which are unused or referenced only once? This may not be an error, but might point to misspellings.

3. Are the attributes which the compiler assigns to each variable the ones you expected?

4. Did the compiler produce any `warning´ messages (assuming of course that you have a successful compile)? These messages point to potential problems.

5. Is the program or module sufficiently `robust?´ That is, does it check its input for validity, or can it be killed with a `reasonable´ number?

6. Does the program NOT do something that you expected it would?

7. DEBUGGING

Your program has finally `compiled´ and `go´ed, but produces erroneous output. This is caused by `bugs´ which must be found and exterminated in order for the program to be `correct.´*

This chapter provides an approach to bug eradication; don´t forget the checklist in the previous chapter. Many of the following suggestions are from [Myer79].

7.1 Think

The building size necessary to contain a computer with the power of your brain would exceed that of the Empire State Building. You may not be able to calculate quickly, but you can intuit. You should be able to debug most of your programs without going near a computer.

7.2 If You Reach An Impasse

- sleep on it. Your subconscious mind has great potential for working things out while you´re doing something else. This is not an excuse to catch a few z´s on the job.

- describe the problem to someone else. By making an effort to tell a good listener what your program is doing, you may discover the problem yourself. The listener need not be an expert on what you´re doing, either; an ignorant person can often see the naked emperor.

7.3 Where There Is One Bug

there are likely to be more. Examine the immediate vicinity of a bug for other errors, since bugs usually result from a misunderstanding of what the program is supposed to do. Note also that small pieces of code may be veritable `roach hotels;´ there are such things as error-prone modules.

7.4 Fix The Error

Don´t just fix the symptom of the error. Make sure that all occurrences of the error are fixed, not just this `just-discovered´ one.

7.5 Bebugging

Recognize that fixing a program is likely to introduce new bugs, because the entire concept of the program may not be fresh in your mind.

7.6 Faulty Bug Repair

Your `fix´ may be wrong! The probability of a correct fix decreases with the size of the program, and varies with the size of the fix. Bug repair should place you mentally in the program design stage.

* If you would rather wave your hand and say, `Yeah, I know it doesn´t work for such-and-such a case, but so what!?´ then the bug is known as a `feature.´

7.7 Use Debugging Tools

only as a second resort. An `interactive debugger´ requires that you learn another language, and only gives a static picture of what a program is doing. Use them as an adjunct to, but not a replacement for, thought.

7.8 Avoid Experimentation

Experiment only as a last resort. A common mistake made by novices is to attempt to solve the problem by changing the program. For example, "Hmmmm...I don´t know what´s wrong, so let´s change this DO loop and see what happens." This kind of behavior has little chance of addressing the actual problem, and muddies the waters by introducing other errors.

7.9 Use an Octal or Hexadecimal Dump

never.

8. THE LANGUAGES

The second most important requirement of computer programs is that they be legible to humans.* We use programming languages when we talk with computers because English is imprecise (although we humans can understand it even when it is misused!). While we maintain precision in our programming, our computer talk must also be understandable by humans, because we are the race which must `maintain´ the programs. This understandability is controlled first, by the language itself, and second, by the way in which we use the language.

The languages are presented in the order of *decreasing* readability. Thus, if you wish your programs to be readable, you should rather write in ratfor, Pascal, or C, and avoid BASIC and programmable calculators. My viewpoint is: *the language defines the machine*. Hence, I may write programs for a `Pascal computer´ or a `FORTRAN Engine´ and not concern myself with what the machine *really* is.

An aside re `standard´ languages: there are no such things! Although a document may exist which describes the `standard´ for a particular language, each implementor includes the bells and whistles which make his own hardware hum. So there are actually several `versions´ of a standard language; moving from one machine to another can still be a difficult proposition.

8.1 Languages Included

ratfor — Invented by Kernighan and Plauger [Kern76] to provide reasonable constructs for FORTRAN, rational fortran is really a preprocessor to the FORTRAN language. It allows all the features of FORTRAN to be accessed, so the advantages of portability, universality, and relative efficiency of FORTRAN are retained. In addition, it has a `macro´ expansion capability, is free-format, and allows comments on the same line as program text. This language is easily read by programmers with a FORTRAN background.

ratfor is maintained in two versions: a source version in ratfor itself (to permit maintenance in a reasonable language) and a source version in FORTRAN66 to allow easy portability amongst computers.

Pascal — A simple language invented by Niklaus Wirth [Jens74] specifically to teach programming, it has found wide use on microcomputers as well as on mainframes. Its structure `enforces´ good programming practices, and catches many errors early in the programming process, saving debugging time and cost. It is free-format, and allows extreme flexibility in data structuring, commenting, and recursion in both data and procedures. The best texts for learning Pascal are [Wirt73], [Jens74], [Atki80], and [Coop82].

C — This is the language in which the UNIX operating system is written. It is a free-format, terse, language whose form contributed much to the style of ratfor. Several significant scientific programs have been written in C. The standard C manual is the reference [Kern78].

* Readability supersedes all program requirements except correctness.

FORTRAN77 The most recent revision of FORTRAN. The major updates it provides to FORTRAN66 are: the CHARACTER type, string processing, the block IF-THEN-ELSE, and greater explicit control over Input/Output (I/O). Syntax is still somewhat inconsistent; and the meaning of one of the FORTRAN66 constructs has been changed(!).

FORTRAN66 The `standard´ scientific language since 1954. It is the lingua franca of the engineering community. All of the `large´ finite element programs have been written in FORTRAN66, and almost all computer vendors offer a FORTRAN66 compiler. Syntax of this language is unusual, and it forces you to write your program backwards from the way you would express it in English. But `everyone´ knows it, so its anomalies are generally overlooked or unrecognized. It is to your advantage to write in some other language if possible; `ratfor´ is a reasonable alternative.

BASIC The most widespread microcomputer language available today. Invented to teach computing to Dartmouth students, this language is feature-poor, and also forces you to program backwards from your thinking processes.

8.2 Languages Excluded

Ada* This language is now under development. It is the new `standard´ DoD language for embedded systems (systems which are part of some other machine, such as a tank, a ship, a torpedo, etc.). Ada is an `enhanced derivative´ of Pascal.

Algol This family of languages preceded the development of Pascal. It is the native language of Burroughs machines.

APL A very powerful, very cryptic, language. The information content of APL programs is extremely dense. Per [Kell81], APL is "so compacted that the source code can be freely disseminated without revealing the programmer´s intent or jeopardizing proprietary rights." It is impossible to maintain programs written in APL; one throws them away and redoes them.

Assembly Assembly language is a mnemonic method for addressing the computer´s `instruction set.´ Usually, it is a very primitive language (note, however, that the Burroughs machine instructions are Algol, a `high level´ language). Programming in assembly language is undesirable because it takes too much time to do it, and because it is nigh impossible to move it to another vendor´s machine. Certainly, some pieces of some programs should be written in assembly language (data acquisition programs and arcade games, for example), but because it can take ten times more human time to write assembly than some other higher-order language, it is rare that one can recover an investment in assembly language code development. The code produced by optimizing compilers is as good as, and in many cases, better than, the code one could produce `by hand´ using assembly language. Therefore, assembly

* Ada is a trademark of the Department of Defense.

language programming is to be eschewed.

COBOL A language designed primarily for business purposes, and not very useful for engineering purposes.

FORTH A `threaded´ language in some use on microcomputers, which forces the programmer to act as part of the compiler. This language immortalizes reverse Polish notation (alias `Okie code´) in programs; FORTH programs are hard to read.

Lisp The de facto language of the artificial intelligence community. It is not designed for, and hence has seen little use in, number crunching.

Modula-2 Niklaus Wirth´s successor to Pascal. It removes many of the difficulties associated with program development in Pascal, but distribution of this language has not been wide-spread to date.

PL/I IBM´s answer to the FORTRAN/COBOL dichotomy. An enormous and complex language. Never really caught on in the engineering community.

TI59 The Texas Instruments´ model 59 Programmable calculator (as well as other models by Texas Instruments, Hewlett-Packard, Sharp, etc.) allows programs to be written in terms of keystrokes. The languages are very close to assembly language; hence, the remarks directed at assembly languages are also applicable here. Although I planned to include examples of the six constructs expressed in this language, they were so unreadable that I omitted them. Because of the recent advances in miniaturization of electronic components, shirt-pocket computers are now available (with BASIC interpreters) for the price of a calculator. Programmable calculators are antiques (which is not to say that they are not useful!).

others Languages which are either dead, distasteful, experimental, unknown to me, or little used for engineering work. This includes languages like MAD, CLU, Gypsy, SNOBOL, Euclid, Simula, RPG, Simscript, etc., etc.

9. RATFOR

Blocks of statements in ratfor are grouped with braces `{´ and `}´. On CDC systems, these may be entered as either braces or brackets `[ ]´. Unfortunately, the CDC character set is not common across all the printers, so brackets may appear as either brackets or braces.

ratfor is a free-format language. It is a good idea to indent your code so that the logical structure of the program is reflected by the layout of the text on a page.

There are three ways to provide commentary in ratfor programs: with a `C´ or `*´ in column 1, or with a `#´ anywhere. You should start your program text in column 2 or 3 so that a `C´ or a `*´ in column 1 is not interpreted as a comment line. Blank lines are ignored by ratfor, so you may also use these to provide additional `white space.´

9.1 Variable Names

Names of variables are limited by the limitations of the target FORTRAN (since ratfor preprocesses text to FORTRAN). Normally, this is a maximum of six characters. Every variable should be declared to be of a specific type, and its purpose defined. This may be done easily through the type definitions REAL, INTEGER, LOGICAL, CHARACTER, COMPLEX, and DOUBLE, and the on-the-same-line commentary ratfor provides. It is inadvisable to rely on FORTRAN to automatically type your variables for you.

The type definitions may also be used to declare the dimensions of arrays, so the DIMENSION statement is not needed in ratfor or FORTRAN programs.

ratfor uses the `reserved word´ concept; i.e., there are several words which are special to ratfor. Thus, in ratfor programs, you may not use any of the following words for variable names: break, do, else, if, next, repeat, until, and while.

9.2 Boolean Expression

X and Y are assumed to be variables of the same type.

X is greater than Y:

```
X > Y
```

X is greater than or equal to Y:

```
X >= Y  or  X => Y
```

X is equal to Y:

```
X == Y
```

X is not equal to Y:

```
X ^= Y (on CDC)
X != Y (on UNIX)
```

X is less than Y:

```
X < Y
```

X is less than or equal to Y:

```
X <= Y  or  X =< Y
```

9.3 Assignment Statement

```
X = expression
```

(Same as FORTRAN.)

9.4 Conditional

```
if (B) {
   S1
   S2
   } # end if
```

9.5 Alternative

```
if (B) {
   S1
   S2
   }
else {
   S3
   S4
   } # end if
```

9.6 Multiple Choice

The clearest way to select from multiple possibilities is with a series of If-Then-Else´s.

```
if ( B1 )
   S1
else if ( B2 )
   S2
else if ( B3 )
   S3
   ...
else
   Sn
```

9.7 While

```
while (B) {
   S1
   S2
   } # end while
```

9.8 Repeat-Until

```
repeat { # until (B)
   S1
   S2
   } until (B)
```

where it is helpful to a reader to have the termination condition of the loop presented in commentary on the Repeat line.

9.9 Determinate Iteration

The ratfor `Do´ is identical to the FORTRAN `Do´, with the exception that ratfor requires no label* as a loop terminator. If the target language is FORTRAN66, then the Do index may only increase; if the target is FORTRAN77, the index may increase or decrease. (Hence, a decreasing index is more clearly handled with a While or a Repeat-Until.)

```
   do I = J,K,M {
      S1
      S2
      } # end do
```

9.10 Altered Loops

9.10.1 Loop Exit

Although any loop (While, Repeat-Until, Determinate) may be `exit´ed, we illustrate with a While.

```
   while (B1) {
      S1
      if (B2)
         break
      S2
      } # end while
```

9.10.2 Loop Redo

Although any loop (While, Repeat-Until, Determinate) may be `re-do´ed, we illustrate with a While.

```
   while (B1) {
      S1
      if (B2)
         next
      S2
      } # end while
```

* In general, the only labels you need in a ratfor program are those on FORMAT statements.

9.11 Generalized Iteration

ratfor provides a generalized iteration scheme called the For. It has the form

```
   for ( initialize; B; reinitialize ) {
      S1
      S2
      }
```

which is the equivalent of

```
   initialize
   while (B) {
      S1
      S2
      reinitialize
      }
```

The For may be easier to understand in some circumstances because it keeps all terms which control the loop on a single `line.´

9.12 ratfor Summary

```
Conditional:       if (B)
                      S

Alternative:       if (B1)
                      S1
                   else
                      S2

Multiple Choice:   if B1
                      S1
                   else if B2
                      S2
                   else if B3
                      S3
                   else if B4
                      S4

While:             while (B)
                      S

Repeat-Until:      repeat
                      S
                   until (B)

Determinate:       do i = j,k,m
                      S

Generalized loop:  for ( init; B; re-init )
                      S

Loop Exit:         break

Loop Redo:         next
```

9.13 Program Header

The ratfor program header requires a comment line whioh identifies the program, followed by the same information as that required in a FORTRAN program. On CDC machines, this has the form

```
# name - a line to demonstrate the header format
  program name( file1, file2, ... )

declaration of variable types
```

9.14 Running a ratfor Program

Assume you have composed your program with a text editor on a CDC computer and the code is in <p>. Assume further that you have already executed the commands

```
ATTACH,ROTH,CCLLIB,ID=CSPR.
LIBRARY,ROTH.
```

(you must execute these commands only once). Then the program in <p> may be run with the command

```
rr,p.
```

(see [Roth83]).

9.15 Examples

I will assume that ratfor is preprocessing programs for FORTRAN66.

9.15.1 Temperature Conversion

```
# ftoc - fahrenheit to celsius conversion, -40f to 100f
  program main(output)
  real f,c    # fahrenheit, celsius temperatures

  f = -40.0
  while( f <= 100.0 ) {
    c = 5.0*(f - 32.0)/9.0
    print *, f,c
    f = f + 5.0
    }
  end
```

9.15.2 Nearest Points Assume that the three coordinates of each point are typed on a single line of <input>.

```
# near - find 2 of 10 points which are closest neighbors
  program near( input, output)
  real x(10,3)       # array of points
  real oldd,newd     # distances between points
  integer i,j,k,m,n  # counters

  read *,((x(i,j),j=1,3),i=1,10) # free format read
                                 # first distance: from 1 to 2
  oldd = sqrt( (x(1,1)-x(2,1))**2
             + (x(1,2)-x(2,2))**2
             + (x(1,3)-x(2,3))**2 )

  do i = 1, 9 {
    ip1 = i + 1
    do j = ip1, 10 {
      newd = sqrt( (x(i,1)-x(j,1))**2
                 + (x(i,2)-x(j,2))**2
                 + (x(i,3)-x(j,3))**2 )
      if( newd <= oldd ) {  # found a new nearest neighbor
        m = i
        n = j
        oldd = newd
        }
      }  # end do j
    } # end do i
  print *,' closest points are at '
  print *, (x(m,i),i=1,3), (x(n,i),i=1,3)
  end
```

9.15.3 Count the `A´s Approach: assume that each line has at most 132 characters, and read an entire line at a time. Also assume that character variables can be represented as integers (this works on CDC machines). If the line is shorter than 132 characters, FORTRAN will act as if the nonexistent characters were blanks. If the line is longer than 132 characters, the excess will be (silently) truncated (perhaps not the best solution).

```
# counta - count the number of times `A´ appears in <input>
  program counta( input,output,tape5=input )
  integer count, line(132), i
  integer eof  # end-of-file function (cdc supplied)

  count = 0
  read(5,1) (line(i),i=1,132)
1 format(132a1)
  while( eof(5) == 0 ) {
    do i = 1, 132
      if( line(i) == ´A´ )
        count = count + 1
    read(5,1) (line(i),i=1,132)
    }
  print *,count
  end
```

10. PASCAL

Blocks of statements in Pascal are grouped with the symbols `begin´ and `end´; you might think of them as `fat brackets.´

Pascal is a free-format language. That is, you may begin your code in any column. It is a good idea to indent your code so that the logical structure of the program is reflected by the layout of the text on a page.

The symbol `(*´ or `{´ opens a comment, and the next occurrence of the symbol `*)´ or `}´ closes a comment.* Comments may occur anywhere and may extend across line boundaries. Blank lines are ignored by Pascal, so you may also use these to provide additional `white space.´

10.1 Variable Names

As in English, Pascal does not limit the number of characters in a variable name. Most implementations of the language only guarantee that the first 8 characters are significant. That is,

```
   afairlylongname
   afairlylongnamewithstuffontheend
```

probably refer to the same variable. Although it is possible to use any number of characters, limit yourself to at most 12.

Note that Pascal forces you to declare the types of your variables before you use them. So while you are declaring their types, you might just as well insert some commentary which declares their purposes.

Pascal uses the `reserved word´ concept; i.e., many words are special to Pascal. Thus, in Pascal programs, you may not name a variable `if,´ `while,´ `repeat,´ etc. A full list of the Pascal reserved words is given in [Jens74].

* The braces are not available on CDC.

10.2 Boolean Expression

X and Y are variables of the same type.

X is greater than Y:

```
X > Y
```

X is greater than or equal to Y:

```
X >= Y
```

X is equal to Y:

```
X = Y
```

X is not equal to Y:

```
X <> Y
```

X is less than Y:

```
X < Y
```

X is less than or equal to Y:

```
X <= Y
```

10.3 Assignment Statement

```
X := expression
```

10.4 Conditional

```
if B then begin
   S1;
   S2
   end;
```

The semicolon is used in Pascal to join statements. `begin' and `end' are `reserved words' which serve the same function as brackets in ratfor.

10.5 Alternative

```
if B then begin
   S1;
   S2
   end
else begin
   S3;
   S4
   end (*if*);
```

10.6 Multiple Choice

There are two ways of selecting from multiple alternatives: the case and the if ... else if constructs.

10.6.1 The Case

```
case I of
   L1: S1;
   L2: S2;
   L3: S3;
      ...
   Ln: Sn
   end (*case*);
```

The labels `L1´, `L2´, `L3´, ..., `Ln´ are the legitimate values which `I´ can assume.

10.6.2 if ... else if

```
if B1 then
  S1
else if B2 then
  S2
else if B3 then
  S3
else if B4 then
  S4(* end if *);
```

10.7 While

```
   while B do begin
      S1;
      S2
      end(*while*);
```

10.8 Repeat-Until

```
   repeat (* until B *)
      S1;
      S2
   until B;
```

This construct is made clearer to a reader by putting the termination condition in commentary on the Repeat line,

10.9 Determinate Iteration

10.9.1 Increasing Index

```
   for I := J to K do begin
      S1;
      S2
      end(*for*);
```

10.9.2 Decreasing Index

```
   for I := J downto K do begin
      S1;
      S2
      end(*for*);
```

In these constructs, J is the initial value of I, and K is the terminal value. Note that, if the initial condition satisfies the test for termination, the loop will not be executed. In Pascal, the increment to a loop may be +1 or -1. No other values are possible. For increments other than +1 or -1, use a While or Repeat-Until construct.

10.10 Alterec Loops

10.10.1 Loop Exit Although any loop (While, Repeat-Until, Determinate) may be `exit´ed, we illustrate with a While.

```
label 13;
   ...

while B1 do begin
   S1;
   if B2 then goto 13;
   S2
   end(*while*);

13: (*continue program*)
```

Although this construct is possible, and indeed works, most Pascal-ers would not recommend it. Clearer, and more in keeping with the style of the language would be something like

```
keepon := true; (* comment explaining `keepon´ *)
while B1 and keepon do begin
   S1;
   if B2 then
      keepon := false
   else begin
      S2
      end
   end(*while*);
```

because it presents all the loop terminators in a single statement (the While statement) and it presents the logic in explicitly logical terms rather than in terms of transportation (`goto label´).

10.10.2 Loop Redo Although any loop (While, Repeat-Until, Determinace) may be `re-do´ed, we illustrate with a While,

```
   label 13;
      ...

   while B1 do begin
      S1;
      if B2 then goto 13;
      S2
   13: end(*while*);
```

Although this construct is also possible, and indeed works, I do not recommend it. Rather, the following is preferred;

```
   while B1 do begin
      S1;
      if B2 then begin
        (* empty `begin end´; a do-nothing! *)
         end
      else begin
         S2
         end
      end(*while*);
```

10.11 Pascal Summary

```
Conditional:       if B then
                      S;

Alternative:       if B1 then
                      S1
                   else
                      S2;

Multiple Choice:   case I of                            if B1 then
                     B1: S1;                               S1
                     B2: S2;            (or)            else if B2 then
                     B3: S3;                               S2
                     B4: S4                             else if B3 then
                   end;                                    S3
                                                        else if B4 then
                                                           S4;
While:             while B
                      S;

Repeat-Until:      repeat
                      S
                   until B;

Determinate:       for I := J {to|downto} K do
                      S;

Loop exit:         additional loop control variable

Loop redo:         additional Conditional construct
```

10.12 Program Header

The Pascal program header requires the following information:

```
   program name( file1, file2, ... );

   label declarations (if any)

   constant declarations (if any)

   type definitions (if any)

   variable definitions (if any)

   begin
```

The program is closed with an `end´ which matches the opening `begin´, followed by a period. That is, the last symbol in the program is `end.´.

10.13 Running a Pascal Program

Assume you have composed your program with a text editor on a CDC computer and the code is in <p>. Assume further that you have already executed the commands

```
   ATTACH,ROTH,CCLLIB,ID=CSPR.
   LIBRARY,ROTH.
```

(you must execute these commands only once). Then the program in <p> may be run with the commands

```
   pc,p.
   lgo.
```

10.14 Examples

10.14.1 Temperature Conversion

```
program ftoc(output);

(*. ftoc - fahrenheit to celsius conversion, -40f to 100f *)

var f,c: real;  (* fahrenheit, celsius temperatures *)

begin
  f := -40.0;
  while f <= 100.0 do begin
    c := 5.0*(f - 32.0)/9.0;
    writeln(f,c);
    f := f + 5.0
    end (*while*)
end.
```

10.14.2 Nearest Points Assume that the three coordinates of each point are typed on a single line of <input>.

```
program near( input, output);

(*. near - find 2 of 10 points which are closest neighbors *)

var
  x: array[1..10,1..3] of real; (* points in space *)
  oldd,newd: real;              (* distances between points *)
  i,j,k,m,n: integer;           (* counters *)

begin
  for i := 1 to 10 do
    for j := 1 to 3 do
      read(x[i,j]);

                                 (* first distance: from 1 to 2 *)
  oldd := sqrt( sqr(x[1,1]-x[2,1])
              + sqr(x[1,2]-x[2,2])
              + sqr(x[1,3]-x[2,3]) );

  for i := 1 to 9 do
    for j := i+1 to 10 do begin
      newd := sqrt( sqr(x[i,1]-x[j,1])
                  + sqr(x[i,2]-x[j,2])
                  + sqr(x[i,3]-x[j,3]) );
      if newd <= oldd then begin  (* found a new nearest neighbor*)
        m := i;
        n := j;
        oldd := newd
        end (* if *)
      end (* for j *);
  write (' closest points are at ');
  for i := 1 to 3 write(x[m,i]);
  for i := 1 to 3 write(x[n,i]);
  writeln
end.
```

10.14.3 Count the `A´s

```
program counta( input,output );

var ch: char;
    count: integer;

begin
  count := 0;
  while not eof do begin
    read(ch);
    if ch = ´A´ then
      count := count + 1
    end(* while *);
  writeln(count)
end.
```

In Pascal, `eof´ is a Boolean function which tests for `end-of-file.´

11. C

Blocks of statements in C are grouped with braces `{´ and `}´. As of this writing, no C compiler exists for CDC computers; it is a language which lives on many other brands, however. These include DEC´s PDP-11/xx and VAX 11/7xx, IBM machines of various sizes, and many microcomputers.

C is a free-format language. That is, you may begin your code in any column. It is a good idea to indent your code so that the logical structure of the program is reflected by the layout of the text on a page. Blank lines are ignored by C, so you may also use these to provide additional `white space.´

Comments may occur anywhere and may extend across line boundaries. The symbol `/*´ opens a comment, and the next occurrence of the symbol `*/´ closes a comment.

11.1 Variable Names

C uses the `reserved word´ concept; i.e., there are a large number of words which are special to C. Thus, in C programs, you may not name a variable `if,´ `while,´ `break,´ etc. A full list of the C reserved words is given in [Kern78].

Only the first 8 characters in a name are significant. More may be used, however. A practical limit is 12 characters.

11.2 Boolean Expression

X and Y are variables of the same type.

X is greater than Y:

```
X > Y
```

X is greater than or equal to Y:

```
X >= Y
```

X is equal to Y:

```
X == Y
```

X is not equal to Y:

```
X != Y
```

X is less than Y:

```
X < Y
```

X is less than or equal to Y:

```
X <= Y
```

11.3 Assignment Statement

```
X = expression ;
```

Note that in C, statements are terminated with a semicolon, rather than joined as in Pascal (a subtle, but sometimes painful, difference).

11.4 Conditional

```
if (B) {
   S1;
   S2;
   } /* end if */
```

11.5 Alternative

```
if (B) {
   S1;
   S2;
   }
else {
   S3;
   S4;
   } /* end if */
```

11.6 Multiple Choice

There are two ways to select from multiple possibilities: if ... else if's, or the Switch.

11.6.1 if ... else if

```
if ( B1 )
   S1;
else if ( B2 )
   S2;
     else if ( B3 )
        S3;
          ...
                 else
                     Sn;
```

Indenting successive `else-if's is deemed by some to make the program text easier to read.

11.6.2 Switch

```
switch ( I ) {
   case L1: S1;
   case L2: S2;
   case L3: S3;
    ...
   case Ln: Sn;
   default: Sx;
} /*end switch*/
```

Choosing between the `switch' and multiple `if ... else if's is mostly a matter of style, although the switch can be faster in some circumstances.

11.7 While

```
while (B) {
   S1;
   S2;
} /* end while */
```

11.8 Repeat-Until

```
do { /* while B */
   S1;
   S2;
   } while (B);
```

where it is helpful to a reader to have the termination condition of the loop presented in commentary on the Do line.

11.9 Determinate Iteration

C provides a generalized iteration scheme called the For. It has the form

```
for ( initialize; B; reinitialize ) {
   S1;
   S2;
   }
```

which is the equivalent of

```
initialize;
while (B) {
   S1;
   S2;
   reinitialize;
   }
```

Note that the initialization and reinitialization steps need not necessarily be strictly related to counting of discrete program iterations. The For may be easier to understand than a While in some circumstances because it keeps all terms which control a loop on a single `line.'

11.10 Altered Loops

11.10.1 Loop Exit Although any loop (While, Repeat-Until, Determinate) may be `exit´ed, we illustrate with a While.

```
while (B1) {
   S1;
   if (B2)
      break;
   S2;
} /* end while */
```

11.10.2 Loop Redo Although any loop (While, Repeat-Until, Determinate) may be `re-do´ed, we illustrate with a While.

```
while (B1) {
   S1;
   if (B2)
      continue
   S2;
   } /* end while */
```

11.11 C Summary

```
Conditional:      if (B)
                     S;

Alternative:      if (B)
                     S1;
                  else
                     S2;

Multiple Choice:  switch (I) of {                if (B1)
                    case B1: S1;                    S1;
                    case B2: S2;     (or)        else if (B2)
                    case B3: S3;                    S2;
                    case B4: S4;                 else if (B3)
                  }                                 S3;
                                                 else if (B4)
                                                    S4;
While:            while (B)
                     S;

Repeat-Until:     do
                     S
                  while (B);

Determinate:      for(init; B; reinit)
                     S;

Loop exit:        break

Loop redo:        continue
```

11.12 Program Header

C is not yet available on CDC machines. On a UNIX system, the C program header is

```
main ()
{
   declaration of variable types
```

Note that the program must be closed with a right brace `}' to match the opening left brace of the program header.

11.13 Running a C Program

Assume you have composed your program with a text editor on a UNIX system and the code is in <p.c>. Then the program in <p.c> may be run with the commands

```
cc p.c
a.out
```

11.14 Examples

I assume in the following examples that the function `printf´ is available to provide output. Most UNIX systems provide it.

11.14.1 Temperature Conversion

```
/*. ftoc - fahrenheit to celsius conversion, -40f to 100f */

main()
{
  float f,c

  f = -40.0;
  while ( f <= 100.0 ) {
    c = 5.0*(f - 32.0)/9.0;
    printf("%4.0f %6.1f\n",f,c);
    f = f + 5.0;
  }
}
```

11.14.2 Nearest Points In this example, please note that you must provide the function `gtarray' (to read the values into the x array), and that the `sqrt' function must be made available to your program by accessing the appropriate system library. C doesn't have an exponentiation operator, either, so you must either multiply the terms yourself (as I've done here) or provide a function to do it.

```
/* near - find 2 of 10 points which are closest neighbors */
main ()
{
  float x[10][3];       /* array of points */
  float oldd,newd;      /* distances between points */
  int i,j,k,m,n;        /* counters */

  gtarray(x,10,3); /* function to fill array `x';
                      you must provide this per
                      your operating system reqts. */

                   /* first distance: from 1 to 2 */
  oldd = 0.0;
  for(k = 1; k <= 3; ++k )
    oldd = oldd + (x[1][k]-x[2][k])*(x[1][k]-x[2][k]);
  oldd = sqrt(oldd);

  for(i = 1; i <= 9; ++i ) {

    for(j = i+1; j <= 10; ++j ) {

      newd = 0.0;
      for(k = 1; k <= 3; ++k )
        newd = newd + (x[i][k]-x[j][k])*(x[i][k]-x[j][k]);
      newd = sqrt(newd);

      if( newd <= oldd ) {   /* found a new nearest neighbor */
        m = i;
        n = j;
        oldd = newd;
      }
    }

  }
  printf(" closest points are at \n");
  printf ("%d %d %d\n",x[m][1],x[m][2],x[m][3]);
  printf ("%d %d %d\n",x[n][1],x[n][2],x[n][3]);
}
```

11.14.3 Count the `A's

```
/*. counta - count the `A's in input. */
main()
{
  int c,na;

  na = 0;
  while( (c=getchar()) != EOF )
    if( c == 'A' )
      ++na;
  printf("%d\n",na)
}
```

where `getchar' is a function which gets the next character from <input> and puts it into the variable `c.' If <input> is at end of file, then `getchar' assigns a non-character value to c (a system dependent constant of 0 or -1). I avoid this issue by using the symbolic EOF.

12. FORTRAN77

Blocks of statements in FORTRAN77 are grouped only in the `if ... then else´ statement, since blocks are a new concept in FORTRAN. It is advisable to `create´ blocks of statements by using the FORTRAN77 do-nothing CONTINUE statement.

FORTRAN77 requires that the text of a program be contained in columns 7 through 72. That is, you may begin your code in any column after column 6. It is a good idea to indent your code so that the logical structure of the program is reflected by the layout of the text on a page.

Comments are provided in a FORTRAN77 program by putting a `C´ or `*´ in column 1.

Since all The Constructs are not available directly in FORTRAN77, you may find it easier to use ratfor than this language.

12.1 Variable Names

Names of variables are limited to a maximum of six characters. Every variable should be declared to be of a specific type and its purpose defined. This may be done easily through the type definitions REAL, INTEGER, LOGICAL, CHARACTER, COMPLEX, and DOUBLEPRECISION. It is inadvisable to allow FORTRAN to type your variables.

The type definitions may also be used to declare the dimensions of arrays, so the DIMENSION statement is not needed in FORTRAN programs (see, for example, the program in Section 12.14.2).

12.2 Boolean Expression

X and Y are variables of the same type.

X is greater than Y:

```
X .GT. Y
```

X is greater than or equal to Y:

```
X .GE. Y
```

X is equal to Y:

```
X .EQ. Y
```

X is not equal to Y:

```
X .NE. Y
```

X is less than Y:

```
X .LT. Y
```

X is less than or equal to Y:

```
X .LE. Y
```

12.3 Assignment Statement

```
X = expression
```

FORTRAN77 statements end at the end of a line. Long statements may be continued for up to 19 additional lines by providing a non-blank, non-zero character in column 6 of the continuation lines (columns 1 through 5 must be blank). Note however, that long FORTRAN77 statements are like long sentences: they're hard to read, hard to understand, and hard to correct! Strive for short statements, even if it means inventing new variables to contain the results of intermediate calculations.

12.4 Conditional

```
if( B ) then
   S1
   S2
endif
```

12.5 Alternative

```
if ( B ) then
   S1
   S2
else
   S3
   S4
endif
```

12.6 Multiple Choice

```
if ( B1 ) then
   S1
else if ( B2 ) then
   S2
else if ( B3 ) then
   S3
 ...
else
   Sn
endif
```

12.7 While

This construct must be simulated. Because a reader may not immediately recognize the construct, it should be annotated as a While. Note that FORTRAN77 forces you to express the Constructs in terms of transportation - "GOTO label" is a rather indirect way to express a logical concept.

```
C                              while B
23000 if ( B ) then
            S1
            S2
            goto 23000
        endif
C                              end while
```

12.8 Repeat-Until

This construct must be simulated. Because a reader may not immediately recognize the construct, it should be annotated as a Repeat-Until.

```
C                              repeat until B
23000 continue
            S1
            S2
            if ( B ) goto 23000
```

12.9 Determinate Iteration

```
        do 23000 I = J,K,M
            S1
            S2
23000       continue
```

J is the initial value of I, K is the terminal value of I, and M is the non-zero increment of I. Note that if the initial condition satisfies the terminal condition, the loop will not be executed.

12.10 Altered Loops

12.10.1 Loop Exit

Although any loop (While, Repeat-Until, Determinate) may be `exit´ed, we illustrate with a While.

```
23000 if ( B1 ) then
         S1
         if ( B2 ) goto 23001
         S2
         goto 23000
      endif
23001 continue
```

12.10.2 Loop Redo

Although any loop (While, Repeat-Until, Determinate) may be `re-do´ed, we illustrate with a While.

```
23000 if ( B1 ) then
         S1
         if ( B2 ) goto 23000
         S2
         goto 23000
      endif
```

12.11 FORTRAN77 Summary

Conditional:

```
if (B) then
   S
endif
```

Alternative:

```
if (B) then
   S1
else
   S2
endif
```

Multiple Choice:

```
if ( B1 ) then
   S1
else if ( B2 ) then
   S2
else if ( B3 ) then
   S3
 ...
else
   Sn
endif
```

While:

```
23000 if ( B ) then
         S1
         S2
         goto 23000
      endif
```

Repeat-Until:

```
23000 continue
        S1
        S2
        if ( B ) goto 23000
```

Determinate:

```
      do 23000 I = J,K,M
        S1
        S2
23000   continue
```

Loop Exit:

```
goto label
```

Loop Redo:

```
goto label
```

12.12 Program Header

The program header on CDC machines has the form

```
program name( file1, file2, ... )
declaration of variable types
```

12.13 Running a FORTRAN77 Program

Assume you have composed your program with a text editor on a CDC computer and the code is in <p>. Assume further that you have already executed the commands

```
ATTACH,ROTH,CCLLIB,ID=CSPR.
LIBRARY,ROTH.
```

(you must execute these commands only once). Then the program in <p> may be run with the command

```
vc,p.
abs.
```

(see [Roth83]).

12.14 Examples

12.14.1 Temperature Conversion

```
      PROGRAM FTOC
C
C.   FTOC - FAHRENHEIT TO CELSIUS CONVERSION, -40F TO 100F
C
C                                 F = FAHRENHEIT, C = CELSIUS
      REAL F,C
C
      F = -40.0
C                                 WHILE( F <= 100 )
1     IF(F .LE. 100.0) THEN
         C = 5.0*(F - 32.0)/9.0
         PRINT *,F,C
         F = F + 5.0
         GOTO 1
      ENDIF
C                                 END WHILE
      END
```

12.14.2 Nearest Points Assume that the three coordinates of each point, are typed on a single line of <input>.

```
      PROGRAM NEAR( INPUT, OUTPUT )
C
C. NEAR - FIND 2 OF 10 POINTS WHICH ARE CLOSEST NEIGHBORS
C
C X(10,3) IS THE ARRAY OF POINTS
C OLDD,NEWD ARE DISTANCES BETWEEN POINTS
C I,J,K,M,N ARE COUNTERS
C
      REAL X(10,3)
      REAL OLDD,NEWD
      INTEGER I,J,K,M,N
C
      READ *,((X(I,J),J=1,3),I=1,10)
C                              FIRST DISTANCE: 1 TO 2
      OLDD = SQRT( (X(1,1)-X(2,1))**2
     $           + (X(1,2)-X(2,2))**2
     $           + (X(1,3)-X(2,3))**2 )
C
      DO 2  I = 1, 9
        IP1 = I + 1
        DO 1 J = IP1, 10
          NEWD = SQRT( (X(I,1)-X(J,1))**2
     $               + (X(I,2)-X(J,2))**2
     $               + (X(I,3)-X(J,3))**2 )
C                              NEW MINIMUM FOUND?
          IF( NEWD .LE. OLDD ) THEN
            M = I
            N = J
            OLDD = NEWD
          ENDIF
1       CONTINUE
2     CONTINUE
      PRINT *,' CLOSEST POINTS ARE AT '
      PRINT *, (X(M,I),I=1,3), (X(N,I),I=1,3)
      END
```

12.14.3 Count the `A's Approach: assume that each line has at most 132 characters, and read an entire line at a time. Also assume that character variables can be represented as integers (this works on CDC machines). If the line is shorter than 132 characters, FORTRAN will act as if the nonexistent characters were blanks. If the line is longer than 132 characters, the excess will be (silently) truncated (perhaps not the best solution).

```
      PROGRAM COUNTA
C
C. COUNT THE NUMBER OF `A'S IN <INPUT>.
C
      INTEGER COUNT,I
      CHARACTER LINE *132
C
      COUNT = 0
      OPEN(UNIT=5,FILE='INPUT')
C                                 WHILE NOT END OF FILE
1     READ(5,2,END=3) LINE
2       FORMAT(A132)
        DO 4 I = 1, 132
           IF(LINE(I:I) .EQ. 'A') THEN
             COUNT = COUNT + 1
           ENDIF
4       CONTINUE
        GOTO 1
C                                 END WHILE
3     CONTINUE
      PRINT *, COUNT
      END
```

13. FORTRAN66

If there is any way to avoid writing programs in this language, you should take it. One way which comes readily to mind is the alternative `ratfor,´ which produces FORTRAN66 but allows you to express your program in terms more nearly like English. Note that the straight-forward application of FORTRAN66 costs you time and effort because you must convolute your natural thought processes when you write the program, and then again when you go looking for errors.

The concept of `blocks of statements´ does not exist in FORTRAN66. It is advisable to `create´ blocks of statements by using the FORTRAN66 do-nothing CONTINUE statement.

FORTRAN66 requires that the text of a program be contained in columns 7 through 72. That is, you may begin your code in any column after column 6. It is a good idea to indent your code so that the logical structure of the program is reflected by the layout of text on a page.

Comments are provided in a FORTRAN66 program by putting a `C´ in column 1.

13.1 Variable Names

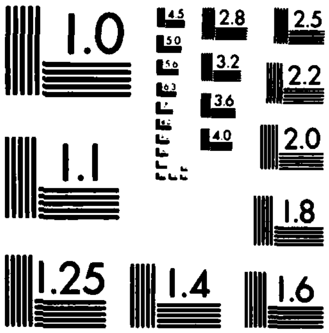
Names of variables are limited to a maximum of six characters. Every variable should be declared to be of a specific type and its purpose defined. This may be done easily through the type definitions REAL, INTEGER, LOGICAL, COMPLEX, DOUBLEPRECISION. It is inadvisable to let FORTRAN tvpe your variables for vou.

The type definitions may also be used to declare the dimensions of arrays, so the DIMENSION statement is not needed in FORTRAN programs (see, for example, the program in section 13.14.2).

AD-A147 964   FUNDAMENTALS OF COMPUTER PROGRAMMING FOR ENGINEERS(U)   2/2
DAVID W TAYLOR NAVAL SHIP RESEARCH AND DEVELOPMENT
CENTER BETHESDA MD P N ROTH OCT 84 DTNSRDC-84/062

UNCLASSIFIED   F/G 5/9   NL

END



MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

13.2 Boolean Expression

X and Y are variables of the same type.

X is greater than Y:

```
X .GT. Y
```

X is greater than or equal to Y:

```
X .GE. Y
```

X is equal to Y:

```
X .EQ. Y
```

X is not equal to Y:

```
X .NE. Y
```

X is less than Y:

```
X .LT. Y
```

X is less than or equal to Y:

```
X .LE. Y
```

13.3 Assignment Statement

```
X = expression
```

FORTRAN66 statements end at the end of a line. Long statements may be continued for up to 19 additional lines by providing a non-blank, non-zero character in column 6 of the continuation lines (columns 1 through 5 must be blank). Note however, that long FORTRAN66 statements are like long sentences: they're hard to read, hard to understand, and hard to correct! Strive for short statements, even if it means inventing new variables to contain the results of intermediate calculations.

13.4 Conditional

In general, FORTRAN66 requires a `reversal of thought´ when expressing conditional tests because the only control available to group statements is the GOTO. Hence, to control several statements, the expression `if B then´ is expressed `if NOT B goto´. The language also forces you to express logic in terms of transportation - "GOTO label" is a rather indirect way to express a logical concept. Compare the following with the definition of the Conditional Construct in Section 4.4 and in some of the other languages: sections 9.4, 10.4, and 11.4.

```
      if(.not.(B))goto 23000
         S1
         S2
23000    continue
```

13.5 Alternative

```
      if(.not.(B))goto 23000
C                        B IS TRUE
         S1
         S2
         goto 23001
23000 continue
C                        B IS FALSE
         S3
         S4
23001 continue
```

Commentary in FORTRAN66 are lines with the symbol `C´ in column 1. Because the control structures are more difficult to understand in this language, they should be more thoroughly commented than in other languages.

13.6 Multiple Choice

This form of multiple choice makes you assign a statement label and a condition number to any of the choices you wish to make.

```
      goto ( 1,2,3,...,n ), I
1     S1
      goto m
2     S2
      goto m
3     S3
      goto m
        ...
n     Sn
m     continue
```

13.7 While

This construct must be simulated. Because someone who reads your program may not immediately recognize the construct, it should be annotated as a While.

```
C                            WHILE B DO
23000 if(.not.(B))goto 23001
         S1
         S2
         goto 23000
23001 continue
```

13.8 Repeat-Until

This construct must be simulated. Because a reader may not immediately recognize the construct, it should be annotated as a Repeat-Until.

```
C                            REPEAT UNTIL (B)
23000 continue
         S1
         S2
23001    if( .not.(B) )goto 23000
```

13.9 Determinate Iteration

13.9.1 Increasing Index

```
        do 23000 I = J,K,M
           S1
           S2
   23000   continue
```

J is the initial value of I, K is the terminal value of I, and M is the positive increment of I. Note that this loop will be executed at least once (like the Repeat-Until).

13.9.2 Decreasing Index A severe lack in FORTRAN66, this may be simulated with the While loop. Old FORTRANers may prefer to simulate this with a Repeat-Until so that the loop executes at least once and is an exact complement of the Do.

```
   C                            LOOP FROM J DOWNTO K BY M
         I=J
   23000 if(I.lt.K)goto 23002
            S1
            S2
            I=I-M
            goto 23000
   23002 continue
```

13.10 Altered Loops

13.10.1 Loop Exit

Although any loop (While, Repeat-Until, Determinate) may be `exit´ed,* we illustrate with a While.

```
23000 if(.not.(B1))goto 23001
         S1
         if( B2 ) goto 23001
         S2
         goto 23000
23001 continue
```

13.10.2 Loop Redo

Although any loop (While, Repeat-Until, Determinate) may be `re-do´ed, we illustrate with a While.

```
23000 if(.not.(B1))goto 23001
         S1
         if( B2 ) goto 23000
         S2
         goto 23000
23001 continue
```

* You can GOTO almost anywhere in a FORTRAN66 program!

13.11 FORTRAN66 Summary

```
Conditional:                if(.not.(B))goto 23000
                               S1
                               S2
                  23000        continue

Alternative:                if(.not.(B))goto 23000
                               S1
                               S2
                               goto 23001
                  23000 continue
                               S3
                               S4
                  23001 continue

Multiple Choice:            goto ( 1,2,3,...,n ), I
                  1         S1
                            goto m
                  2         S2
                            goto m
                  3         S3
                            goto m
                              ...
                  n         Sn
                  m         continue

While:            23000 if(.not.(B))goto 23001
                              S1
                              S2
                              goto 23000
                  23001 continue

Repeat-Until:     23000 continue
                              S1
                              S2
                              if ( .not.(B) ) goto 23000

Determinate:                do 23000 I = J,K,M
                              S1
                              S2
                  23000       continue

Loop Exit:                  goto label

Loop Redo:                  goto label
```

13.12 Program Header

The FORTRAN66 program header on CDC machines has the form

```
program name( file1, file2, ... )
declaration of variable types
```

13.13 Running a FORTRAN66 Program

Assume you have composed your program with a text editor on a CDC computer and the code is in <p>. Assume further that you have already executed the commands

```
ATTACH,ROTH,CCLLIB,ID=CSPR.
LIBRARY,ROTH.
```

(you must execute these commands only once). Then the program in <p> may be run with the command

```
fc,p.
abs.
```

(see [Roth83]).

13.14 Examples

13.14.1 Temperature Conversion

```
      PROGRAM FTOC(OUTPUT)
C
C.   FTOC - FAHRENHEIT TO CELSIUS CONVERSION, -40F TO 100F
C
C                             F = FAHRENHEIT, C = CELSIUS
      REAL F,C
C
      F = -40.0
C                             WHILE F <= 100
1     IF(F .GT. 100.0) GOTO 2
         C = 5.0*(F - 32.0)/9
         PRINT *,F,C
         F = F + 5.0
         GOTO 1
C                             END WHILE
2     CONTINUE
      END
```

13.14.2 Nearest Points Assume that the three coordinates of each point are typed on a single line of <input>.

```
      PROGRAM NEAR( INPUT, OUTPUT )
C
C. NEAR - FIND 2 OF 10 POINTS WHICH ARE CLOSEST NEIGHBORS
C
C X(10,3) IS THE ARRAY OF POINTS
C OLDD,NEWD ARE DISTANCES BETWEEN POINTS
C I,J,K,M,N ARE COUNTERS
C
      REAL X(10,3)
      REAL OLDD,NEWD
      INTEGER I,J,K,M,N
C
      READ *,((X(I,J),J=1,3),I=1,10)
C                               FIRST DISTANCE FROM 1 TO 2
      OLDD = SQRT( (X(1,1)-X(2,1))**2
     $           + (X(1,2)-X(2,2))**2
     $           + (X(1,3)-X(2,3))**2 )
C
      DO 2  I = 1, 9
        IP1 = I + 1
        DO 1 J = IP1, 10
          NEWD = SQRT( (X(I,1)-X(J,1))**2
     $               + (X(I,2)-X(J,2))**2
     $               + (X(I,3)-X(J,3))**2 )
C                               NEW MINIMUM FOUND?
          IF(.NOT.( NEWD .LE. OLDD )) GOTO 3
            M = I
            N = J
            OLDD = NEWD
3           CONTINUE
1         CONTINUE
2       CONTINUE
      PRINT *,' CLOSEST POINTS ARE AT '
      PRINT *, (X(M,I),I=1,3), (X(N,I),I=1,3)
      END
```

13.14.3 Count the `A´s Approach: assume that each line has at most 132 characters, and read an entire line at a time. Also assume that character variables can be represented as integers (this works on CDC machines). If the line is shorter than 132 characters, FORTRAN will act as if the nonexistent characters were blanks. If the line is longer than 132 characters, the excess will be (silently) truncated (perhaps not the best solution).

```
      PROGRAM COUNTA(INPUT,OUTPUT,TAPE5=INPUT)
C
C. COUNT THE NUMBER OF `A´S IN <INPUT>.
C
      INTEGER COUNT, I, LINE(132)
      INTEGER EOF
C
      COUNT = 0
      READ(5,1) (LINE(I),I=1,132)
1     FORMAT(132A1)
C                           WHILE NOT END OF FILE
2     IF( EOF(5) .NE. 0) GOTO 4
       DO 3 I = 1, 132
         IF(LINE(I) .EQ. 'A') COUNT = COUNT + 1
3        CONTINUE
       READ(5,1) (LINE(I),I=1,132)
       GOTO 2
C                           END WHILE
4     CONTINUE
      PRINT *, COUNT
      END
```

where we note that the integer function EOF is a CDCism.

14. BASIC

There are many versions of BASIC. The following remarks therefore apply to BASIC in a general sense; for specifics, you will need the BASIC manual for your system.

The concept of `blocks of statements′ doesn′t exist in BASIC. Most BASICs prohibit the indentation of code to reflect the logical structure of a program; thus BASIC programs are usually difficult to read and write.

Comments are provided in a BASIC program by putting the string `REM′ (for REMark) as the first entry following the line number. Because the control structures are more difficult to understand in this language, they should be more thoroughly commented than in other languages.*

Since few of the Constructs are available directly in BASIC, you may find it easier to use any other language.

14.1 Variable Names

The names of BASIC variables may be one or two characters long. The first character must be a letter, and the second may be a letter or a number. Most BASICs assume all variables are of type REAL (i.e, they have fractional parts). Character type variables usually include the `$′ as the last character in the name ′e.g., N$).

* Unfortunately, BASIC is often run on machines whose memory is too small to hold both code and comments!

14.2 Boolean Expression

X and Y are variables of the same type.

X is greater than Y:

```
X > Y
```

X is greater than or equal to Y:

```
X >= Y
```

X is equal to Y:

```
X = Y
```

X is not equal to Y:

```
X <> Y
```

X is less than Y:

```
X < Y
```

X is less than or equal to Y:

```
X <= Y
```

14.3 Assignment Statement

```
LET X = expression
```

BASIC statements end at the end-of-line. Continuation of statements is often impossible, so you need to simplify expressions by inventing new variables to hold the results of intermediate calculations.

14.4 Conditional

In general, BASIC requires a `reversal of thought´ when expressing conditional tests. This is because the only command available to group statements is the `{goto} linenumber´. In many implementations, the `goto´ is not supplied, but understood. Hence, to control several statements, the expression `if B then´ is expressed `if NOT B linenumber´.

```
120 if not B then 150
130 S1
140 S2
150 rem ...continue
```

14.5 Alternative

```
10 if not B then 50
15 rem                 B IS TRUE
20 S1
30 S2
40 goto 80
50 rem                 B IS FALSE
60 S3
70 S4
80 rem continue
```

14.6 Multiple Choice

The BASIC multiple choice forces you to assign a program line number (in the proper numerical sequence) which corresponds to the condition number of the choice you wish to make. This may be difficult to do when you are designing a program because you can `run out´ of line numbers. Allow `enough´ numbers.

```
10 on I goto ( 20,40,60,90 )
20 S1
30 goto 110
40 S2
50 goto 110
60 S3
70 goto 110
90 S4
100 goto 110
110 rem ...continue
```

14.7 While

This construct must be simulated.

```
 5 rem              WHILE B DO
10 if not B then 50
20 S1
30 S2
40 goto 10
50 rem  continue
```

14.8 Repeat-Until

This construct must be simulated.

```
10 rem              REPEAT UNTIL B
20 S1
30 S2
40 if not B then 10
```

14.9 Determinate Iteration

This construct is handled nicely in many BASICs.

```
10 for I = J to K step M
20 S1
30 S2
40 next I
```

J is the initial value of I, K is the terminal value of I, and M is the non-zero increment of I. Note that if the initial condition satisfies the terminal condition, the loop will not be executed.

14.10 Altered Loops

14.10.1 Loop Exits

Although any loop (While, Repeat-Until, Determinate) may be `exit´ed (you can GOTO just about anywhere in a BASIC program), we illustrate with a While.

```
10 if not B1 then 60
20 S1
30 if B2 then 60
40 S2
50 goto 10
60 rem continue
```

14.10.2 Loop Redo

Although any loop (While, Repeat-Until, Determinate) may be `re-do´ed, we illustrate with a While.

```
10 if not B1 then 60
20 S1
30 if B2 then 10
40 S2
50 goto 10
60 rem  continue
```

14.11 BASIC Summary

```
Conditional:          10 if not B then 30
                      20 S
                      30 ...

Alternative:          10 if not B then 40
                      20 S1
                      30 goto 50
                      40 S2
                      50 ...

Multiple Choice:      10 on I goto ( 20,40,60,...,n )
                      20 S1
                      30 goto m
                      40 S2
                      50 goto m
                           ...
                       n Sn
                       m ...

While:                10 if not B then 40
                      20 S
                      30 goto 10
                      40 ...

Repeat-Until:         10 S
                      20 if not B then 10

Determinate:          10 for I = J to K step M
                      20 S
                      30 next I

Loop Exit:               goto linenumber

Loop Redo:               goto linenumber
```

14.12 Program Header

BASIC programs do not require a header on most systems. However, it is best to use at least one comment line to identify the program.

14.13 Running a BASIC Program

Please refer to the manual for the system you have. BASIC programs are usually entered from a keyboard or recalled from auxiliary storage (tape or floppy disk). The sequence to run a program is usually something like

```
RUN programname
```

14.14 Examples

14.14.1 Temperature Conversion

```
5 REM FAHRENHEIT TO CELSIUS CONVERSION, -40F TO 100F
10 FOR F = -40 TO 100 STEP 5
20 LET C = 5*(F - 32)/9
30 PRINT F,C
40 NEXT F
```

14.14.2 Nearest Points Assume that the three coordinates of each point are contained in DATA statements (some points are provided below). It is usually easier to use the editing capabilities of the BASIC interpreter by supplying DATA statements than it is to correctly type in 30 numbers `interactively.´

```
10 REM NEAR - FIND 2 OF 10 POINTS WHICH ARE CLOSEST NEIGHBORS
20 REM X(10,3) IS THE ARRAY OF POINTS
30 REM C,D ARE DISTANCES BETWEEN POINTS
40 REM I,J,K,M,N ARE COUNTERS
50 DIM X(10,3)
65 REM ... GET POINTS FROM DATA STATEMENTS, LINES 280-370
70 FOR I = 1 TO 10
80 FOR J = 1 TO 3
90 READ X(I,J)
100 NEXT J
110 NEXT I
111 REM ... FIRST DISTANCE FROM 1 TO 2
112 C = 0
113 FOR K = 1 TO 3
114 C = C + (X(1,K)-X(2,K))^2
115 NEXT K
116 C = SQRT(C)
118 REM ... BEGIN LOOKING AT NEIGHBORS
120 FOR I = 1 TO 9
130 IP1 = I + 1
140 FOR J = IP1 TO 10
150 D = 0
160 FOR K = 1 TO 3
170 D = D + (X(I,K)-X(J,K))^2
180 NEXT K
190 D = SQRT(D)
195 REM ... FOUND NEW MINIMUM?
200 IF D > C THEN 240
210 M = I
220 N = J
230 C = D
240 NEXT J
245 REM ... END FOR J = IP1 TO 10
250 NEXT I
255 REM ... END FOR I = 1 TO 9
260 PRINT " CLOSEST POINTS ARE AT ";
270 PRINT X(M,1),X(M,2),X(M,3),X(N,1),X(N,2),X(N,3)
280 DATA 10,20,30
290 DATA 21,31,41
300 DATA 32,42,52
310 DATA 10.3,11,27
320 DATA -27.006,23,4
330 DATA 22,9.3,26
340 DATA 1,2,2.1
350 DATA 3,31,20
360 DATA 3,31.1,-24
370 DATA 19,-21.07,3.001
380 END
```

14.14.3 Count the `A's The BASIC language recognizes the character data type. However, the language does not have a standard way to read a file; data are usually embedded in the program itself. To access files on a system running BASIC, one must somehow execute calls to the operating system through `extensions' to the language. Since these calls vary widely from vendor to vendor, I omit this example.

15. ACKNOWLEDGMENTS

My ideas about programming, pedagogy, and people have been honed by continual arguments with Mel Haas, Jim McKee, Dave Sommer, Kevin Laurent, Sharon Good, and Fredd Thrasher. I thank them for the good influence they have had on me and on this work.

Tony Cincotta provided insight in formatting C code for legibility, and corrected several errors in my first attempts at writing the chapter on C.

Stefan Zilliacus, Stan Wilner, Steve Walter, Joyce Toler, Claude Rizzo, Bill Mynatt, and Fran Fortin pointed to portions of this text which were irrelevant, obscure, inconsistent, or wrong, and helped me to make this document clearer and smaller.

Many other people have had an impact on this document. Although I have not mentioned them by name (the list would be longer than this text!), I wish them to please accept my thanks.

16. REFERENCES

Atki80 Atkinson, Laurence, Pascal Programming, John Wiley & Sons (1980).

Berz71 Berztiss, A T, Data Structures Theory and Practice, Academic Press (1971).

Boud71 Boudreau, Ace, "Peeling the Yellow Tomato," Journal of the Oriental Ecdysiast Society, Vol 17, No. 3 (1971).

Carr96 Carroll, Lewis, Through the Looking Glass, (1896).

Coop82 Cooper, Doug and Michael Clancy, Oh! Pascal!, W W Norton & Co (1982).

Day72 Day, A Colin, FORTRAN Techniques with Special Reference to Non-Numerical Applications, Cambridge University Press (1972).

Jens74 Jensen, K and Wirth, W, Pascal User Manual and Report, Springer-Verlag (1978).

Kell81 Kelly-Bootle, Stan, The Devil's DP Dictionary, McGraw-Hill (1981).

Kern76 Kernighan, B W and P J Plauger, Software Tools, Addison-Wesley (1976).

Kern78 Kernighan, B W & Ritchie, D W, The C Programming Language, Prentice Hall (1978).

Myer79 Myers, Glenford J, The Art of Software Testing, John Wiley & Sons (1979).

Roth82 Roth, Peter N, THE STRUCTURES DEPARTMENT INTERACTIVE CDC PRIMER, Enclosure to ltr Ser 82-175-2 of 20 Jan 82.

Roth83 Roth, Peter N, THE PROCEDURE BOOK: A Guide to Engineers and Scientists Using CDC Computers, David Taylor Naval Ship Research and Development Center Technical Memorandum 83-(1703).2-63, June (1983).

Wirt73 Wirth, Niklaus, Systematic Programming: An Introduction, Prentice-Hall (1973).

Wirt76 Wirth, Niklaus, Algorithms + Data = Programs, Prentice-Hall (1976).

17. GLOSSARY

algorithm
a description of the steps necessary to perform a calculation. This is not to be confused with algorasm, which is what a programmer has the first time his program runs.

code
programs, or fragments of programs, are called `code' because they represent ideas in a form which can be read by a machine.

compiler
a program which translates some `high-level' language (such as Pascal, FORTRAN, Ada) into the equivalent instructions that a computer can understand.

computer
an extremely gullible machine which often does exactly what you tell it to do, rather than what you really want it to do.

down
the state in which a computer is immune to user input.

English
an obtuse language correctly spoken only by Edwin Newman and William F Buckley, Jr.

error
something which causes a program to not compile (sometimes called a typographical error), or which causes a program to be `not running.' Note that a program which never stops is considered to be a `not running' program.

execution
jargon for `run.'

function
(a.) what a program is supposed to do. (b.) a program module (e.g., the TANGENT function).

go
verb describing the act of attempting to get a program to run.

implementation
how a program is put on a particular computer system.

interactive
In an `interactive' computing environment, man and machine have a dialog: man types command, machine does the work, man types command, etc. As opposed to `batch,' in which man perforates regularly shaped pieces of pasteboard, man totes pasteboards to machine, machine looks through holes, machine prints what it mis-read, man totes pasteboard and paper back to perforator for another cycle.

library
a special kind of file in which one may store programs, subroutines, functions, and the like. Libraries make it easy for you to use code developed by others.

line
(a.) telephone connection to a computer. (b.) a series of characters terminated by a `line-feed' character. (c.) the entire contents of a computer punchcard.

GLOSSARY

load
a CDC verb. To `load´ a program is to put a machine language program into memory along with other modules from other libraries. On other systems, this is known as `binding´ or `linking´ or `link-editing.´

loop
a synonym for iteration.

macro
a fragment of code which is `expanded´ by a program called a macro-processor. Macros are used to extend the power of a language. Simple examples: the arithmetic statement function of FORTRAN66 and FORTRAN77, and the PARAMETER statement of FORTRAN77.

memory
is the place where the computer `stores´ data. Because reading the computer memory doesn't destroy what is read, one can perceive that the computer `remembers´ numbers. Memory is often called `prime store´ in British publications.

operating system
a computer program which `runs the machine´ and allows other programs to run.

preprocessor
a program which runs before the `real´ work starts. For example, a text editor is a preprocessor to ratfor, ratfor is a preprocessor to FORTRAN, FORTRAN is a preprocessor to the LOADER, the LOADER is a preprocessor to the SYSTEM, the SYSTEM is a preprocessor (and co-processor) of YOUR PROGRAM.

procedure
(a.) a Pascal subroutine; (b.) a CCL program. (c.) a way of doing things.

program
a set of definitions, declarations, data, and algorithms which becomes a component of a computer.

recursion
the expression of a function in terms of itself. E.g., the factorial n! may be expressed recursively as

$n! = n*(n-1)!$ with $n > 0$ and $1! = 1$.

Of the languages discussed here, recursion is possible only in C and Pascal.

register
a computer component which can hold a data item.

robust
software is said to be robust when it can withstand the assault of any data without `going down.´

run
Execution of a program.

running
Engineering programs are `running´ when they produce correct output. Any other program is `not running.´ See also `error.´

shell
a program which provides the interactive interface between man and modern computers.

software
the hard part of computing (as opposed to hardware, which is the easy part).

store
as a verb: to put data into the computer's memory. as a noun: the place where data are kept. See also ‘memory.’

symbol
a representation of a single thing; a symbol may be composed of more than one character. For example, the Pascal symbols used to group statements are ‘begin’ and ‘end.’

system
the computer program that allows other computer programs to run. Also called ‘executive’ and ‘monitor.’

word
a grouping of information on a machine which is convenient to the manufacturer. On CDC equipment, a word is ten 6-bit characters; on some DEC equipment, a word is four 8-bit bytes.

18. INDEX

INITIAL DISTRIBUTION

Copies	
1	DNA
	1 SPSS
3	CNR
	1 ONR-100
	1 ONR-200
	1 ONR-400
4	NRL
	1 2800
	1 4000
	1 5000
	1 6000
2	NAVMAT
	1 PM 4
	1 PM 23
2	NAVAIR
	1 JP-1
	1 JP-2
3	NAVFAC
	1 04
22	NAVSEA
	1 05
	1 05B
	1 05R
	1 55X
	1 55X1
	1 55X11
	1 55X12
	1 55X13
	1 55Y
	1 55Y2
	3 55Y21
	3 55Y22
	3 55Y23
	1 63R
	1 PMS 402
	1 PMS 406
3	NDAC
	1 NDAC
	1 NDAC 40
	1 NDAC 92

Copies	
2	NUSC
	1 WR
	1 WU
1	NUSC
1	United States Naval Academy
1	NAVWARCOL
1	NOSC
1	NAVSHIPYD BREM/LIB
1	NAVSHIPYD CHASN/LIB
1	NAVSHIPYD MARE/LIB
1	NAVSHIPYD NORVA/LIB
1	NAVSHIPYD PEARL/LIB
1	NAVSHIPYD PHILA/LIB
1	NAVSHIPYD PTSMH/LIB
1	NAVSHIPYD PUGET/LIB
1	SUPSHIP GROTON
1	SUPSHIP NPTNWS
12	DTIC
1	AFFTC-ENCSD
	1 Richard Black STOP 200
2	NBS/INST FOR COMPUTER SCI & TECH
	1 Roger Martin
	1 Dolores Wallace
1	NASA/LANGLEY
	1 GWU CENTER
1	University of Akron/Stow Ohio Steven Arnold
1	MIT Dept of Mech Eng/Dr. K.-J. Bathe

Copies	
1	University of Notre Dame Prof. Michael Katona
1	George Washington Univ School of Eng & Applied Science Prof. H. Leibowitz
1	General Dynamics/Electric Boat/CT System Technology/Structural Programs
4	Weidlinger Associates
1	Lockheed Missiles & Space Co, Inc Dr. David Bushnell
1	Rock Island Arsenal Thomas Crane
1	Grumman Data System Corp. Rolf Gaeding
3	Bolt Beranek & Newman Inc./ New London CT/V. Godino
1	Bell Laboratories Mel Haas
1	Structural Dyn Res Corp Steven G. Harris
1	Texas Instruments/Austin TX Warren Hunt
1	Duke Laboratory Thomas J. Langan
1	United States Geological Survey Kevin Laurent
1	INFOtek Charles Maiorana
1	Ingalls Shipbuilding/Ship Sys Sur/M.S. McGowan
1	Martin Marietta Data System Gene Nutt

Copies	
1	NKF Engineering Assoc/Vienna Dr. M. Pakstys
1	CDC MNA02B Wayne Reynolds
1	BBN Laboratories Inc Gregg W. Schudel

CENTER DISTRIBUTION

Copies	Code	Name
1	004.1	J. Moton
1	004.1	D. Hibbert
1	012.2	B. Nakonechny
1	012.21	J. Wilson
1	04	
1	12	
1	15	
30	1509	
1	1568	Dr. W. McCreight
1	1568	L. Motter
1	16	
1	17	W.W. Murray
1	1702	
1	1703.1	A. Wilner
100	1703.2	P. Roth
3	1706	
30	172	
20	173	
25	174	
1	175	J. Englehardt
1	175	F. Fortin
1	175	M. Hoffman
1	175	S. Walter
1	175	S. Zilliacus
17	175	
12	177	

Copies	Code	Name
1	18	
1	1843	R. Van Eseltine
1	1843	M. Marquardt
1	1890	G. Gray
1	1890	W. Mynatt
20	1892	J. Strickland
1	1892	S. Wilner
1	1892	S. Good
1	1892	D. Sommer
1	1892	L. Minor
1	19	
1	1936	J. Allender
1	27	
1	28	
1	6080	B. Pierce
10	5211	Reports Distribution
1	522.1	TIC (C) + 1 (m)
1	522.1	TIC (A) + 1 (m)
2	5231	Office Service (A)

FILM